University of South Carolina

Scholar Commons

Theses and Dissertations

Spring 2020

# An Overlay Architecture for Pattern Matching

Rasha Elham Karakchi

Follow this and additional works at: https://scholarcommons.sc.edu/etd

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation

Karakchi, R. E.(2020). *An Overlay Architecture for Pattern Matching.* (Doctoral dissertation). Retrieved from https://scholarcommons.sc.edu/etd/5943

This Open Access Dissertation is brought to you by Scholar Commons. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholar Commons. For more information, please contact dillarda@mailbox.sc.edu.

المنارة للاستشارات

www.manaraa.com

# AN OVERLAY ARCHITECTURE FOR PATTERN MATCHING

by

Rasha Elham Karakchi

Bachelor of Science
University of Mosul, 2005

Master of Science
University of Mosul, 2008

Master of Science
University of South Carolina, 2016

---

Submitted in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy in

Computer Science and Engineering

College of Engineering and Computing

University of South Carolina

2020

Accepted by:

Jason D. Bakos, Academic Advisor

Duncan Buell, Committee Member

Manton Matthews, Committee Member

Herbert Ginn, Committee Member

Yan Tong, Committee Member

Cheryl L. Addy, Vice Provost and Dean of the Graduate School

المنارة للاستشارات

www.manaraa.com

© Copyright by Rasha Elham Karakchi, 2020
All Rights Reserved.

المنارة للاستشارات



www.manaraa.com

# Dedication

To my beloved sons, Raghid and Tamer



المنارة للاستشارات
www.manaraa.com

# Acknowledgments

It is the dream of a young woman from a destroyed third-world country in the Middle East, suffered fear, faith persecution, and insecurity. Seven years in first-world country where everything is new, starting from nowhere and stepping through many challenges. With two little sons and husband, living the happiness, unhappiness, success, disappointments, longing to re-unite with the parents, and the fear of the unknown future. Proudly, I'm presenting my dissertation, the outcome of this journey, and it would not have been possible without the contribution of several people.

First, I would like to express my sincere gratitude to my advisor Dr. Jason Bakos, for his exemplary support, patience and guidance throughout my doctoral studies.

I would like to extend my thanks to my dissertation committee: Dr. Duncan Buell, Dr. Manton Matthews, Dr. Herbert Ginn and Dr. Yan Tong, for taking their time to serve on my committee. I would like also to acknowledge the assistance of Charles A. Daniels and Lothrop O. Richards, who helped in implementing some of the design work.

Finally, the source of support, my family. I will forever be grateful to my parents, Elham Karakchi and Saja Suleiman, for their unconditional love and endless encouragement. Thanks should also go to my dear sisters Rana and Maryam for their ever love and care.

With no doubt, the most deserving of my gratitude is the love of my life and my best friend, my husband Alaa Jameel, who has always been beside me and constantly encouraged me in every step of this journey. His unlimited support has ever sourced me the power to overcome all life obstacles.




المنارة للاستشارات
www.manaraa.com

# Abstract

Deterministic and Non-deterministic Finite Automata (DFA and NFA) comprise the fundamental unit of work for many emerging big data applications, motivating recent efforts to develop Domain-Specific Architectures (DSAs) to exploit fine-grain parallelism available in automata workloads.

This dissertation presents NAPOLY (Non-Deterministic Automata Processor OverLaY), an overlay architecture and associated software that attempt to maximally exploit on-chip memory parallelism for NFA evaluation. In order to avoid an upper bound in NFA size that commonly affects prior efforts, NAPOLY is optimized for runtime reconfiguration, allowing for full reconfiguration in 10s of microseconds. NAPOLY is also parameterizable, allowing for offline generation of repertoire of overlay configurations with various trade-offs between state capacity and transition capacity.

In this dissertation, we evaluate NAPOLY on automata applications packaged in ANMLZoo benchmarks using our proposed state mapping heuristic and off-shelf SAT solver. We compare NAPOLY's performance against existing CPU and GPU implementations. The results show NAPOLY performs best for larger benchmarks with more active states and high report frequency. NAPOLY outperforms in 10 out of 12 benchmark suite to the best of state-of-the-art CPU and GPU implementations. To the best of our knowledge, this is the first example of a runtime-reprogrammable FPGA-based automata processor overlay.



المنارة للاستشارات
www.manaraa.com

# Table of Contents

Dedication . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . iii

Acknowledgments . . . . . . . . . . . . . . . . . . . . . . . . . . . . iv

Abstract . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . v

List of Tables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ix

List of Figures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . x

Chapter 1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . 1

Chapter 2 Background . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.1 Finite Automata . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.2 Micron Automata Processor . . . . . . . . . . . . . . . . . . . . 8

2.3 Automata on ALTERA-FPGA . . . . . . . . . . . . . . . . . . . . 9

2.4 VASIM Relaxation . . . . . . . . . . . . . . . . . . . . . . . . . 12

2.5 ANMLZoo Benchmarks . . . . . . . . . . . . . . . . . . . . . . . 13

Chapter 3 Related Work . . . . . . . . . . . . . . . . . . . . . . . . 15



المنارة للاستشارات
www.manaraa.com

3.1 Synthesis NFAs and Regular Expressions . . . . . . . . . . . . . . . . 15

3.2 Mapping Applications to AP Execution Model . . . . . . . . . . . . . 16

3.3 Open Source Automata Processor Architectures, Simulations, and Benchmarks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

3.4 Optimization Methods For NFA Descriptions . . . . . . . . . . . . . . 19

3.5 Comparative Studies of NFA Implementations on CPUs, GPUs, and FPGAs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

Chapter 4 NAPOLY Design . . . . . . . . . . . . . . . . . . . . . . . 22

4.1 Overlay Architecture . . . . . . . . . . . . . . . . . . . . . . . . 22

4.2 I/O Interface . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

4.3 NAPOLY Performance Model . . . . . . . . . . . . . . . . . . . . . 28

Chapter 5 Mapping Problem . . . . . . . . . . . . . . . . . . . . . . . 31

5.1 Greedy Mapping Heuristic . . . . . . . . . . . . . . . . . . . . . . 32

5.2 SAT solver mapping algorithm . . . . . . . . . . . . . . . . . . . . 34

5.3 NFA Transformation . . . . . . . . . . . . . . . . . . . . . . . . . 38

Chapter 6 Experimental Analysis . . . . . . . . . . . . . . . . . . . . 41

6.1 Hardware Resources . . . . . . . . . . . . . . . . . . . . . . . . . 41

6.2 NAPOLY Run Time . . . . . . . . . . . . . . . . . . . . . . . . . . 43



المنارة للاستشارات
www.manaraa.com

6.3 Mapping Results . . . . . . . . . . . . . . . . . . . . . . . . . . . 44

6.4 NFA Transformation Results . . . . . . . . . . . . . . . . . . . . . . . 46

6.5 Performance Comparison . . . . . . . . . . . . . . . . . . . . . . . . 49

6.6 Overlay Scalability . . . . . . . . . . . . . . . . . . . . . . . . . . 50

Chapter 7 Conclusion and Future work . . . . . . . . . . . . . . . . 52

7.1 Future Research Directions . . . . . . . . . . . . . . . . . . . . . . . 52

Bibliography . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56

Appendix A Mapping Heuristic . . . . . . . . . . . . . . . . . . . . 60

A.1 validate_edges . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60

A.2 check_move . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60

A.3 Move_STE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61

A.4 calculate_score . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65



للاستشارات المنارة

www.manaraa.com

# List of Tables

Table 2.1 Transition Table for DFA Description . . . . . . . . . . . . . . . . 7

Table 2.2 Transition Table for NFA Description . . . . . . . . . . . . . . . . 7

Table 2.3 ANMLZoo Applications . . . . . . . . . . . . . . . . . . . . . . . . 14

Table 6.1 Hardware Resources Used in Different Overlay Configurations . . . 41

Table 6.2 Total M20K Used for Output Buffer . . . . . . . . . . . . . . . . . 42

Table 6.3 Repertoire of the achieved NAPOLY Configurations . . . . . . . . 43

Table 6.4 NAPOLY Mapping Using Mapping Heuristic . . . . . . . . . . . . 46

Table 6.5 NAPOLY Mapping Using SAT solver . . . . . . . . . . . . . . . . 47

Table 6.6 Snort Transformation Results . . . . . . . . . . . . . . . . . . . . 48

Table 6.7 Protomata Transformation Results . . . . . . . . . . . . . . . . . . 48

Table 6.8 Power En Transformation Results . . . . . . . . . . . . . . . . . . 49

Table 6.9 Performance Results . . . . . . . . . . . . . . . . . . . . . . . . . 49

Table 6.10 Repertoire of Achieved Configurations on Stratix10 GS . . . . . . 50

Table 7.1 Wire Utilization Achieved For ANMLZoo Benchmarks . . . . . . . 53



المنارة للاستشارات

www.manaraa.com

# List of Figures

Figure 2.1 DFA for regular expression pattern “ababc”. . . . . . . . . . . . . . 6

Figure 2.2 NFA for regular expression pattern “ababc”. . . . . . . . . . . . . . 6

Figure 2.3 NFA-ANML for regular expression pattern “ababc”. . . . . . . . . 8

Figure 2.4 Micron-AP STE . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

Figure 2.5 Matching “ababc” by mapping Figure 2.2 directly to Micron AP. 9

Figure 2.6 Hierarchical layout of processing elements in Micron half-chip . . 10

Figure 2.7 Island-Style FPGA Architecture . . . . . . . . . . . . . . . . . . . 10

Figure 2.8 FPGA layers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

Figure 2.9 mapping Figure 2.2 directly to FPGA. . . . . . . . . . . . . . . . . 11

Figure 2.10 An Example of Fan-in based relaxation. . . . . . . . . . . . . . . . 12

Figure 2.11 An example of Fan-out based relaxation. . . . . . . . . . . . . . . . 13

Figure 3.1 An example of partitioning NFA based on colors. . . . . . . . . . 19

Figure 4.1 State Element STE Design . . . . . . . . . . . . . . . . . . . . . . 23

Figure 4.2 An example of NAPOLY interconnects. . . . . . . . . . . . . . . . 24

Figure 4.3 NAPOLY Timing Diagram . . . . . . . . . . . . . . . . . . . . . . 28

Figure 4.4 NAPOLY Output Region. . . . . . . . . . . . . . . . . . . . . . . 30

Figure 5.1 Mapping Problem. . . . . . . . . . . . . . . . . . . . . . . . . . . 32

Figure 5.2 An example for generating CNF clauses of literals based on *Constraint 1*. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

Figure 5.3 Transformation of NFA graph in Figure 5.1. . . . . . . . . . . . . 39



للاستشارات المنارة
www.manaraa.com

Figure 5.4 Speeding up performance versus replications. . . . . . . . . . . . . . 40

Figure 6.1 Execution time makeup of NAPOLY. . . . . . . . . . . . . . . . . 44

Figure 6.2 NAPOLY Performance vs. NFA size. . . . . . . . . . . . . . . . . 45

Figure 6.3 Speedup achieved in 75% of Benchmarks at SAT solver vs. heuristic. 47

Figure 7.1 Suggested NAPOLY design . . . . . . . . . . . . . . . . . . . . . 54

Figure A.1 Remapping STEs. Edge between state "fifth" and "second" is reassigned from STEs $n$ and $m$, where $n > m$, to $m$ and $m + 1$ (after an operation "move STE $n$ to $m$"). In this case, a movement from a higher-numbered STE to a lower-numbered STE causes all other STEs assignments between the two values to shift up, requiring an update to all other edges involving these STEs. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65



المنارة للاستشارات
www.manaraa.com

# Chapter 1

# Introduction

Datasets comprised of symbolic data, such as genomic sequences, item sets, graph edges, web data, biological data, and data packets, are growing rapidly in size and computational requirements. Computing such data often involves pattern matching based computation, for example when discovering motifs [29], de novo assembling [25], web-search and ranking [6] , question answering systems [24] [10], compression in NoSQL systems [26] [20], approximate string matching [16], calculating the edit distance between two genomic sequences [34], signature-based threat detection [8], association rule mining [15], and data-packet inspection [8]. Such pattern matching computations are algorithmically reducible to the simulation of either Determinitic and Non-deterministic Finite Automata (DFA and NFA).

A DFA is inherently sequential because only one state is active at a time, and must contain one state that corresponds to every possible partial match of every pattern to be accepted. An NFA allows an arbitrary number of active states, which contains more parallelism, however NFAs are data intensive. The state transition tables of the NFA scales at $O(n^2)$ in the worst case, where $n = \#states$ whose access pattern is data dependent. Access to the state transition tables has parallelism that scales with the number of active states. Thus, evaluating automata on general-purpose architectures becomes limited at large pattern set as a result of the massive inherently unpredictable memory access pattern of transition table.

The inefficient performance of large-scale instance of DFA and NFA on general-purpose architectures as a result of memory bound and unpredictable memory access



المنارة للاستشارات
www.manaraa.com

is driving interest in Domain-Specific Architectures (DSA) to exploit this parallelism without becoming bottleneck on the state transition tables. DSAs for automata processing are generally based on Multiple-Instruction Single-Date (MISD) architecture to perform multiple pattern matching operations on single input data in parallel.

Evaluating Finite Automata on hardware-based applications comprises of two main steps: reconfiguration step and pattern matching step. The reconfiguration step represents loading automata on target platform, loading input symbols into buffer, and flushing reports into buffer. Loading automata can be executed by configuring the RAM [5] or synthesizing automata directly on hardware fabric [23], [40], [3], [22]. The pattern matching step is when the input sequences are streamed on automata to find a match.

Prior works have focused on optimizing the pattern matching time (processing time) with neglecting reconfiguration time, which has significant impact especially when pattern sets are large and multiple reconfiguration times are needed to program the chip, or when many matches are reported at same time. Such automata-based architectures are classified as FPGA-based architectures and ASIC-based architectures.

Field-Programmable Gate Arrays (FPGA)-based approaches, where automata is synthesized directly to an FPGA fabric, have long re-configuration time (10s of seconds) [23], [40], [3], [22], but achieve very high pattern matching throughput (100s of MB/s).

Application Specific Integerated Circuit (ASIC) architectures such as Micron Automata Processor (Micron AP), where the data to be searched is streamed into multiple functional units, where each functional unit tracks partial pattern matches. Such designs have faster re-configuration time than FPGA-based approaches, but lack the ability to make tradeoffs between state density and transition density [5] [22].



المنارة للاستشارات
www.manaraa.com

The main objective of this dissertation is to maintain the performance of automata processing for arbitrarly large pattern sets by implementing an automata overlay on FPGA, Non-Deterministic Automata Processor OverLay (NAPOLY), for pattern matching that can be reconfigured rapidly at runtime and processed at high speed.

NAPOLY achieves as a compromise between purely FPGA- and ASIC-based approaches, allowing for rapid runetime reconfiguration while still having architectural customization. NAPOLY is designed as Processor-in-Memory application, which exploits as much on-chip memory bandwidth allowed by the target automata while supporting arbitrary large automata workloads. The work is comprised into three main contributions:

1. A paramertizable overlay, **NAPOLY**, which is comprised of an array of hardware modules (called State Transition Element or STE), each sensitive to a specific pattern and reconfigured at run time in 21 to 74 $\mu$s depending on the overlay size selected.

2. An open-source tool, **NFATOOL**, which is created principally to map logical pattern states on STEs, the physical entities comprising overlay, using heuristic allocation algorithm. NFATOOL is also developed for SAT solver-based mapping by generating the Conjunctive Normal Form (CNF) files and run off-shelf SAT solver. NFATOOL is extended to transform automata files into different forms such as graph description language, such as DOT files for graph visualization and Transaction Control Language, such as DO files for simulation purposes.

3. Tradeoff, Performance Comparisons and Scalability: (1) An analysis of the tradeoff between state capacity, interconnect density, output buffer size, and reconfiguration time, (2) performance comparison to state-of-the-arts Intel's



المنارة للاستشارات
www.manaraa.com

CPU-based NFA software (Hyperscan) and a well-known GPU-based implementation (iNFAnt) [17], and (3) performance and scalabilty on larger FPGAs.

This dissertation is organized as follows: in Chapter 2, we provide some background information on Finita Automata, Micron Automata Processor (Micron AP), Field-Programmable Gate Array (FPGA), NFA transformation and a brief description about the benchmarks we used for testing and evaluating NAPOLY. In Chapter 3, we present history of related works, techniques and methodologies used to improve automata processing. In addition to, several designs are implemented to exploit the FPGA overlay benefits. Chapter 4 represents the hardware part of the dissertations, where NAPOLY overlay design and its output, and input constraints are described, as well as NAPOLY runtime behavior. NFATOOL is described in Chapter 5, where the mapping heuristics algorithm, SAT solver and NFA Transformation are explained. Chapter 6 presents the experimental analyses, trade-offs, and the results. Finally, Chapter 7 concludes the research outcomes and the future works.



للاستشارات المنارة
www.manaraa.com

# Chapter 2

# Background

This chapter presents some background information regarding Finite Automata and its forms. We then provide an overview on Automata Processing on Domain-Specific Architectures, namely FPGA and Micron AP . Also, we provide some context information on FPGAs, and its architecture. Finally, we explain the NFA Transformation and its approaches, in addition to presenting Automata-based applications Benchmarks (ANMLZoo).

## 2.1 Finite Automata

Finite Automata (FA) is a set of states connected by labeled-edges, which are driven by the input sequence, finite automata M as defined by [27].

**Definition 1.** $M = (Q, \sum, \delta, q_0, F)$, where

- Q is finite set of states,
- $\sum$ is a finite set of symbols called the input alphabet,
- $\delta : \begin{cases} Qx\sum \rightarrow Q, & \text{Transition Function for DFA,} \\ Qx(\sum \cup \lambda) \rightarrow 2^Q, & \text{Transition Function for NFA,} \end{cases}$
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is a set of reporting states.

At each clock cycle, automata makes a transition based on (1) current state activation and (2) the match of input symbol and the edge label. When match is found



المنارة للاستشارات
www.manaraa.com

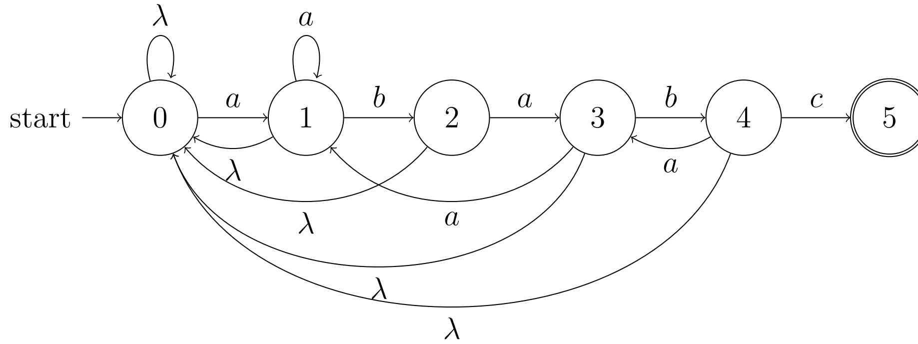



Figure 2.1 DFA for regular expression pattern "ababc".

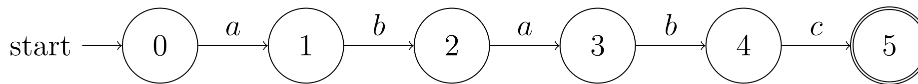



Figure 2.2 NFA for regular expression pattern "ababc".

and automata is driven to a "report" state, automata accepts. In this case, both the report ID and the current symbol position (offset) in the input sequence are reported.

FA is classified either as Deterministic (DFA) and Finite Automata (NFA), which is commonly used to implement regular expression and to design sequential digital logic. However, the two forms operate differently, which lead to different demands on underlying hardware.

### 2.1.1 Deterministic Finite Automata (DFA)

During operation, DFA may have one active state at any time and accesses only one entry of its state transition table and thus must contain a state for every possible partial match of every possible pattern. This can lead to combinatorial growth of the state space, size of state transition table, and consequently a tremendous storage capacity required. Figure 2.1 shows an example of DFA consisting of 6 states (state 0 represents start-state and state 5 represents report-state) that recognizes a simple regular expression pattern "ababc" .



www.manaraa.com

Table 2.1 Transition Table for DFA Description

| Current State | Input Symbol | Next State |
|---|---|---|
| 0 | a | 1 |
| 0 | $\lambda$ | 0 |
| 1 | b | 2 |
| 1 | a | 1 |
| 1 | $\lambda$ | 0 |
| 2 | a | 3 |
| 2 | $\lambda$ | 0 |
| 3 | b | 4 |
| 3 | a | 1 |
| 3 | $\lambda$ | 0 |
| 4 | c | 5 |
| 4 | a | 3 |
| 4 | $\lambda$ | 0 |

Table 2.2 Transition Table for NFA Description

| Current State | Input Symbol | Next State |
|---|---|---|
| 0 | a | 1 |
| 1 | b | 2 |
| 2 | a | 3 |
| 3 | b | 4 |
| 4 | c | 5 |

### 2.1.2 Non-Deterministic Finite Automata (NFA)

NFA differs from DFA, where multiple states can simultaneously be active. Each state needs to only track the progress towards accepting one specific pattern instead of all possible patterns, which produces less number of states, smaller state transition table, and minimal memory requirement as compared to an equivalent DFA. Figure 2.2 shows an NFA form that accepts the same pattern as in Figure 2.1. As it is shown in Table 2.2, the current-state table for NFA is 2.6 times smaller than that of the DFA in Table 2.1.

There is an alternative form of NFA description called ANML (Automata Network Markup Language) developed by Micron [5]. ANML-NFA is differentiated by associating the transition labels with the states instead of the edges. This form adds an



المنارة للاستشارات
www.manaraa.com

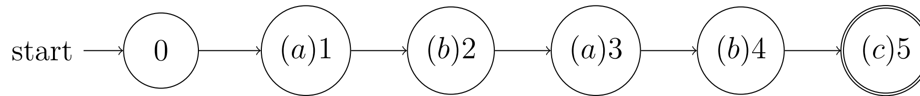



Figure 2.3 NFA-ANML for regular expression pattern “ababc”.

additional constraint that all of each state’s incoming transitions have the same label set, allowing an implementation to associate the current-state table with the states instead of the edges and thus reducing the memory requirement. Figure 2.3 shows the alternative form of NFA with symbols associated with states, for implementing the pattern “ababc”.

## 2.2 Micron Automata Processor

The AP’s architecture inherently requires an ANML-NFA form, where all transitions into each state (from all immediate predecessor states) must activate on the same set of input symbols. Using this form allows the state which is defined as STE to store the symbols associated with the incoming transitions to each state as illustrated in Figure 2.5, which illustrates how automata [ababc] is mapped onto Micron AP. AP STE is shown in Figure 2.4.

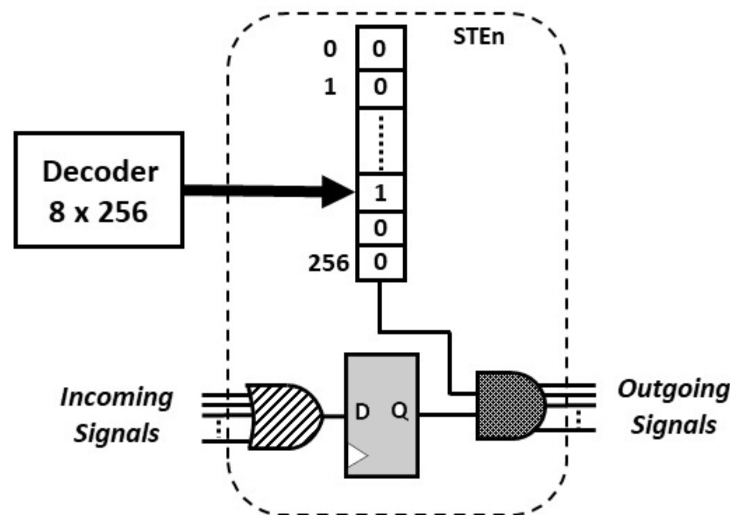



Figure 2.4 Micron-AP STE



المنارة للاستشارات
www.manaraa.com

The AP's routing matrix is built using tri-state switches that can form arbitrary connections between different pairs of STEs using a pool of shared physical wires. The switches and wires are arranged hierarchically, where the interconnectivity between STEs is highest at lower levels of the hierarchy. Starting from the bottom of the hierarchy, there are two STEs to a Group of Two (GoT), eight GoTs to a row (16 STEs), sixteen rows to a block (256 STEs), 96 blocks to a half-chip (24K STEs), and two half-chips to an AP (48K STEs), there are no shared routings (or interconnections) between half-chips. Micron AP adds counters and Boolean elements. First-generation boards have 8 chips distributed on 4 ranks giving 1.5 M STEs total. Figure 2.6 shows the hierarchical layout of processing elements in Micron half-chip.

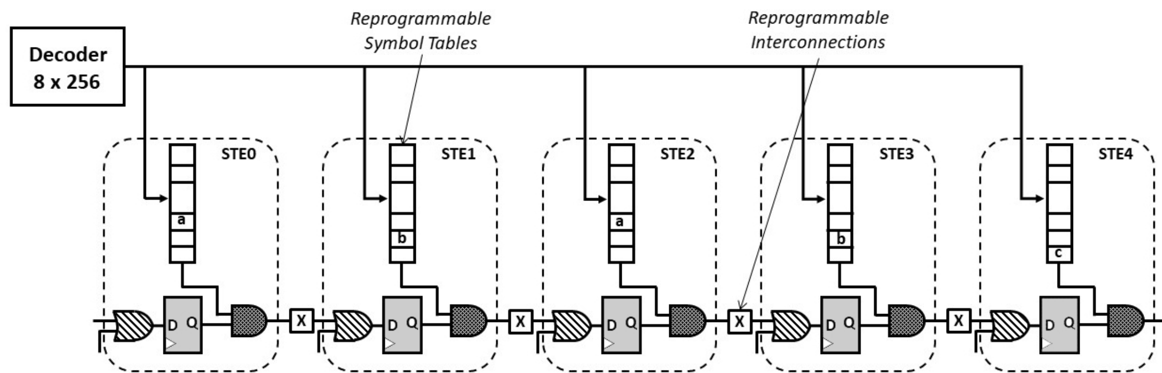



Figure 2.5 Matching "ababc" by mapping Figure 2.2 directly to Micron AP.

## 2.3 Automata on ALTERA-FPGA

Field-Programmable Gate Array (FPGA) is a two-dimensional array of Logic Array Blocks (LABs). Each LAB consists of ten basic reconfigurable Adaptive Logic Modules (ALMs) sharing local interconnections, control signals, and chain of connection lines. The ALM consists of two 6-input Look-Up Table (LUTs), two-adders, four multiplexers, and four registers to implement logic, arithmetic, and register functions. Some LABs are called MLABs (Memory LAB), which contain LUTs-based SRAM capability to support simple dual-port SRAM.



المنارة للاستشارات
www.manaraa.com

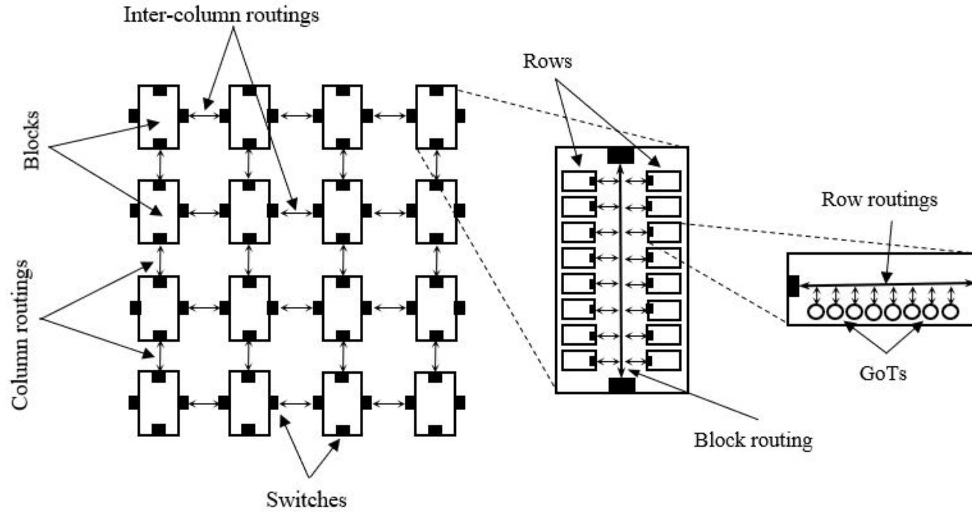



Figure 2.6 Hierarchical layout of processing elements in Micron half-chip

LABs connect to each other through global interconnections distributed horizontally and vertically on the device. Figure 2.7 shows the architecture of ALTERA FPGA.

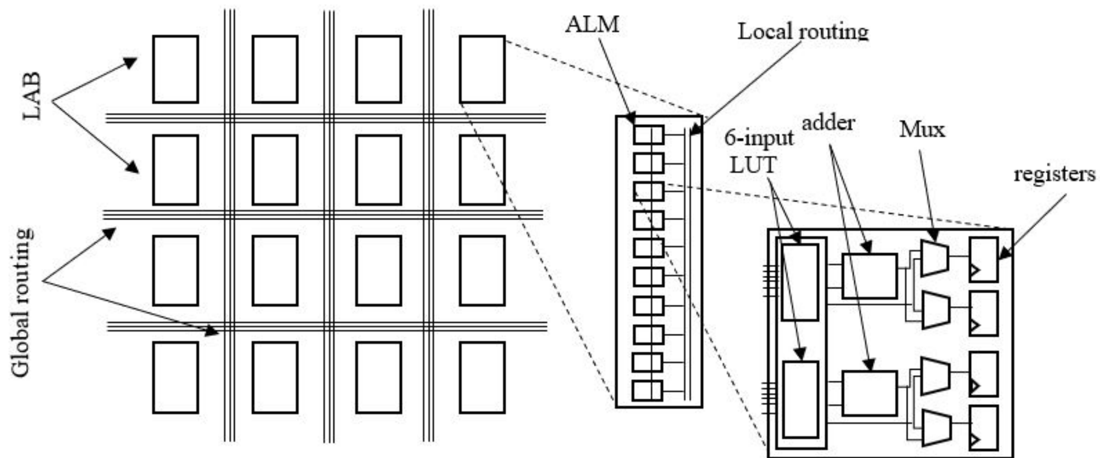



Figure 2.7 Island-Style FPGA Architecture

Conceptually, FPGA can be divided into two layers (as shown in Figure 2.8): ***The logic (or fine-grain layer)***, which represents the pool of hardware resources, and ***The overlay (or configuration layer)***, which consists of SRAM components and defines how the construction of the hardware resources as real circuit.



المنارة للاستشارات
www.manaraa.com

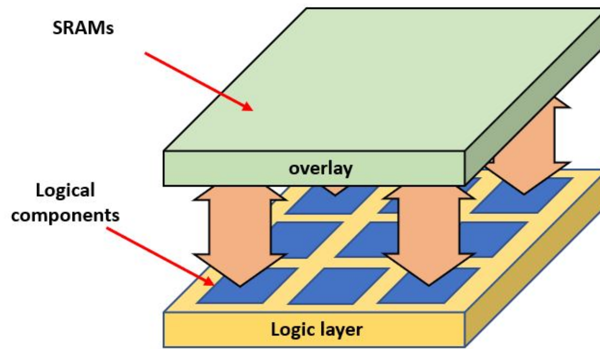



Figure 2.8 FPGA layers

Some prior work [23] [3] implemented automata (or NFA) as combinational circuits and Flip-flops as illustrated in Figure 2.9, where automata [ababc] implemented on FPGA using the traditional approach. This approach has fixed interconnection and fixed symbol tables, which make it inapplicable for Time-Division Multiplexing.

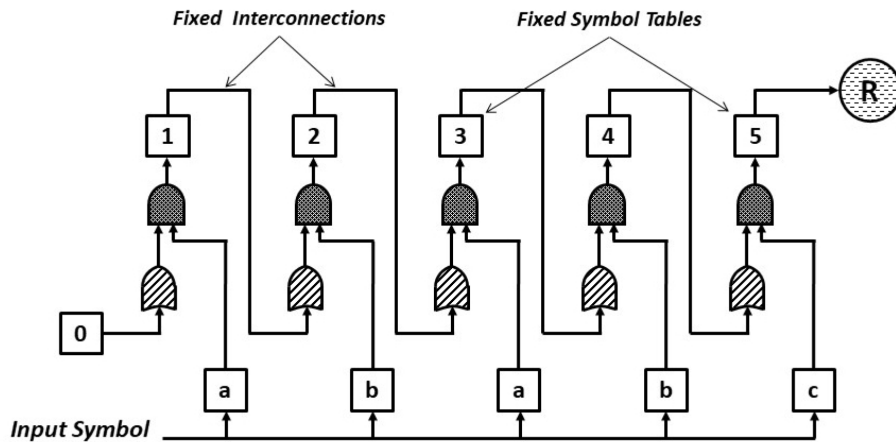



Figure 2.9 mapping Figure 2.2 directly to FPGA.

Developing an abstraction of Automata Processor on FPGA overlay, against which patterns may be synthesized to it is a substantially less costly operation than synthesizing directly to the FPGA fabric.



المنارة للاستشارات
www.manaraa.com

## 2.4 VASIM Relaxation

To allow mapping automata onto hardware platform sometimes requires transforming automata into another functionally equivalent automata, but having different structure. One of the approaches of NFA transformation is VASIM Relaxation [38], which includes two main approaches: Fan-in and Fan-out Relaxations.

**Fan-in Relaxation** is used to transform the NFA which arbitrarily has large fan-in, and violates the hardware fan-in restrictions. A **Fan-in constraint** is the maximum hardware fan-in defined by the architect to allow allocating automata onto hardware platform. The state that violates the Fan-in constraint will be replicated. For illustration, an example is shown below.

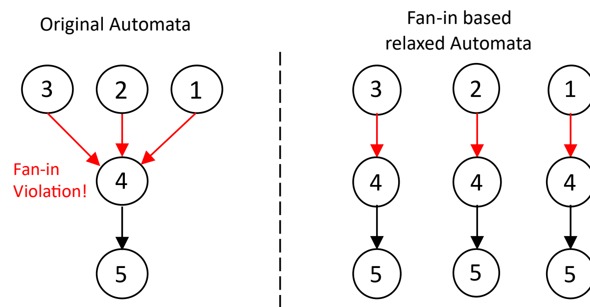



Figure 2.10 An Example of Fan-in based relaxation.

Figure 2.10 shows an original automata of 5 vertices, where its maximum logical fan-in $I$ is 3, and logical fan-in $d$ is limited to 1. Assuming $d$ is 1, state 4 violates the hardware fan-in constraint by its 3 incoming edges. When Fan-in relaxation, the violated state is replicated by *ceil(I/d)*. The outputs of the original vertex are copied, while the inputs are divided among the new replicated vertices.

**Fan-out Relaxation** is the application of **Fan-out constraint**, the maximum hardware fan-out, is defined by the architect to allocate automata onto overlay. Figure 2.11 shows an original automata of 5 vertices, where its maximum logical fan-out $O$ is 3. Assuming the logical fan-out $d$ is limited to 1, state 2 violates the hardware fan-out constraint by its 3 outgoing edges.



www.manaraa.com

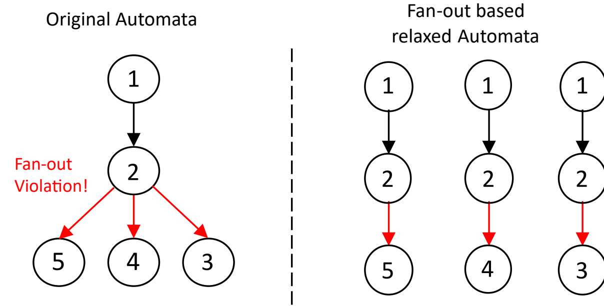



Figure 2.11 An example of Fan-out based relaxation.

During relaxation, the violated state is replicated by *ceil(O/d)*. The outputs of the original vertex are divided among the new replicated vertices, and the inputs are copied.

## 2.5 ANMLZoo Benchmarks

ANMLZoo is a diverse benchmark suite of finite automata for evaluating automata processing engines [11]. It consists of 12 benchmarks representing various applications for automata processing. Table 2.3 shows ANMLZoo benchmarks which have up to 100 thousands states and up to 5000 distinct sub-graphs, which are connected to each other to form ANMLZoo graph. While the first column lists the **Benchmarks**, the second and third columns show the **States** in (1000's) and **Distinct Sub-graphs** respectively for each benchmark. The fourth column shows the Maximum **Logical Fan-in/ Logical Fan-out** for each benchmark, which represents the maximum incoming and outgoing transitions of state. The **Family** column represents the family that each benchmark belongs to. There are three families: Regex (set of characters that define search pattern), Mesh (regular structure with fan-in/fan-out), and widget family (when automata represented as a Tree). Last column **Function** describes the function that each benchmark performs.



www.manaraa.com

Table 2.3 ANMLZoo Applications

| Benchmark | States (K) | Distinct Sub-graphs | Logical Fan-in / Fan-out | Family | Function |
|---|---|---|---|---|---|
| Brill | 26 | 1962 | 4/4 | Regex | brill tag patterns and correct tags |
| ClamAv | 48 | 515 | 11/2 | Regex | viruses signatures in files |
| Snort | 69 | 2585 | 19/5 | Regex | particular snort rules |
| Protomata | 42 | 2340 | 3/106 | Regex | particular motif signature |
| Dotstar | 96 | 2837 | 2/2 | Regex | spy rules |
| Power En | 40 | 2857 | 4/3 | Regex | complex rules |
| Levenshtein | 27 | 24 | 8/5 | Mesh | edit distance between DNA sequence |
| Hamming | 11 | 93 | 4/2 | Mesh | number of mismatches between sequences |
| SPM | 100 | 5025 | 3/2 | Widget | groups of related items |
| Fermi | 40 | 2399 | 2/2 | Widget | particular path |
| Entity Resolution | 95 | 1000 | 28/2 | Widget | input sequences match encoded pattern |
| Random Forest | 75 | 3767 | 2/2 | Widget | Recognize particular handwritten texts |



المنارة للاستشارات
www.manaraa.com

# Chapter 3

# Related Work

In this chapter, we summarize prior work in five related areas: (1) methods for synthesizing automata-type architectures onto an FPGA fabric, (2) applications that benefit from such architectures, (3) open source automata models and architectures, (4) tools and methods for optimizing automata descriptions, (5) comparative studies of NFA implementations on different platforms.

## 3.1 Synthesis NFAs and Regular Expressions

FPGA implementation of regular expression matchers are often inspired by networking applications [41] and many of these are based on automata-based architectures. For these approaches a significant challenge is the high cost of logic synthesis and place-and-route for each set of regular expressions to be implemented.

Yang and Prasanna developed early methods for synthesizing regular expressions into logic mapped onto two specific FPGA devices, a Xilinx Virtex XCV100 (20x30 array of configurable logic blocks) and a conceptual Self-Reconfigurable Gate Array (SRGA) device [30]. Their original approach bypassed the synthesis flow and directly targeted the low-level FPGA fabric. However, as FPGA technology matured this approach became infeasible, and their second design targeted HDL but introduced additional optimization methods for both the NFA descriptions and generated architecture [23] [40].

Becchi et al developed a set of techniques for optimizing both NFA and DFA-based architectures [2] [21] [3], including several approaches to identify and explore design



المنارة للاستشارات
www.manaraa.com

parameters that have the most significant impact on the performance and cost of the corresponding NFA and DFA implementation. Examples of these include alphabet size, number of inputs read per cycle (stride), and storage of next state tables in logic and/or RAM.

Another approach for implementing DFAs and regular expressions is by using Terenary Content-Addressable Memory TCAM, which is specialized type of high-speed memory that searches its entire contents in a single clock cycle. Although it is fast, it lacks of scalability [14].

Teubner et al. [32] implemented an FPGA-based automata engine for database systems by integrating the FPGA hardware as xml projection (or pre-filtering) into database system path. Xml projection technique [19] extracts filtering expressions from query then pre-filtering the data to reduce dataset, and compilation overhead.

### 3.2 Mapping Applications to AP Execution Model

Automata-based architectures are most commonly associated with regular expression evaluation, but the introduction of the Automata Processor has generated interest in identifying other applications that map to NFA-type architectures, or so-called "pattern recognition processors". Examples include association rule mining [39], brill tagging [42] [43], and Levenshtein and Hamming distance calculation [33]. More specific examples include Protomata and Motomata [29], which search for motifs–or common approximate DNA subsequences among a group of genomes–in which each motif is identified by NFA-based pattern loaded onto the AP during runtime. For these, the performance of the AP depends on its ability to quickly synthesize and load patterns onto the AP. Another motif example, Wang et al. [18] proposed an improvement of de novo motif search Weeder performance up to 751x comparing with CPU. There are also efforts to develop general-purpose programming languages for NFA-type architectures, such as RAPID, a proposed high-level programming language



المنارة للاستشارات
www.manaraa.com

for pattern recognition processors [36] . Moreover, Automata Processor is proposed in [9] as an engine to execute integer and floating-point comparison.

## 3.3 Open Source Automata Processor Architectures, Simulations, and Benchmarks

Wadden et al. developed a place and route tool built on VPR [4] that targets a conceptual design for a theoretical Automata Processor fabric [36]. This tool serves as an experimental framework with which to explore the impact of routing algorithms and interconnect design on performance and efficiency. Using this tool, they compared the hierarchical design of the AP routing matrix to a non-hierarchical mesh-based network-on-chip and concluded that the ideal interconnect architecture depends on the input NFA topology.

The same group compiled a suite of NFA benchmarks called ANMLZoo containing a representative example of an NFA description, sample input, and expected outputs for every publicly-released application for the AP as well as two synthetic benchmarks [11]. They also developed open source tool that can simulate the evaluation of arbitrary ANML descriptions and perform basic transformations to NFA such as elimination of counters and Boolean elements and use of state replication to limit the maximum in-degree (fan in) and out-degree (fan out) of the NFA [22]

Fang et al. designed the Unified Automata Processor (UAP), a set of vector extensions added to a traditional von Neuman CPU optimized for implementing a variety of NFA-based programming models [22]. The UAP exploits parallelism by concurrently traversing one edge per cycle for each of its 64 lanes. The design stores NFA transitions in local memory attached to each lane, equally 1 MB in total. The transitions are stored in a compact, efficient format but the design is limited to NFAs that can fit into the local memory.



www.manaraa.com

As a way to exploit the SRAM speed and energy efficiency comparing with DRAM, Das et al. [31] proposed Adopted Micron-AP design, which uses the higher level of cache as a substrate for automata processing instead of DRAM in Micron AP. In the design, conventional cache is extended by two fully pipelined stages to process the input symbol. The first stage represents finding the symbol match in the RAM and the second is implementing the state transitions through hierarchical switching network. Total cache space utilized is 1MB and 12x speed up over AP. However, the growing interconnection complexity with the number of states have limited cache automaton speed and throughput. J et al. [12] uses Time-Division Multiplexing approach by adding multiplexer to pipeline the hierarchical switching network. This approach improved cache automata throughput by 2X.

Wadden and al. [35] proposed a modified Micron AP Reporting Architecture to reduce AP overhead and stall cycles when dense reporting actions occurs at same time. The modified AP reporting region consists of 64 16-bit sub-RA (Reporting Aggregation) equivalent to one 1024-bit RA in Micron AP, all gathered in arbitration unit. Along with reporting aggregation, there is shared 64-bit Mega tag component to report the symbol offset. This architecture improves the reporting sparsity of some ANMLZoo benchmarks and keeps same performance for other benchmarks which have dense of reporting.

Chuncken Bo et al. [7] proposed an automata processing framework implemented using Amazon EC2 F1 Instance. The I/O circuitry of the framework is implemented based on AXI-PCIe combination. The research work shows that FPGA and Micron AP outperform the Von-Neumann architecture due to their massive parallel architecture, however reporting activity majorly impacted the performance to achieve higher clock speed.



المنارة للاستشارات
www.manaraa.com

## 3.4 Optimization Methods For NFA Descriptions

Recent work has contributed new methods for transforming NFA descriptions into alternative but functionally-equivalent forms.To the best of our knowledge, only two research groups has focused on optimizing and transforming automata.

The First approach of NFA transformation is NFA partitioning algorithm, which is proposed by Becchi's group at NC State University [28]. The objective of this algorithm is to split the NFA into a small number of balanced partitions by assigning a unique color to every partition. While state replications increases during splitting and coloring partitions, the number of state replication decreases when consolidating some partitions. The partition size is limited by the hardware platform ($N_{\max}$), where the number of states in each partition must not exceed $N_{\max}$. Figure 3.1 shows NFA partitioning in an example, where the original Automata is partitioned into four separated partitions, represented in the Figure by four unique colors. The green partition comprises of states (0,1,2,3), and pink partition comprises of states (0,4,5,6), blue partition comprises of states (0,4,7) and orange partition comprises of (0,8,9) states.

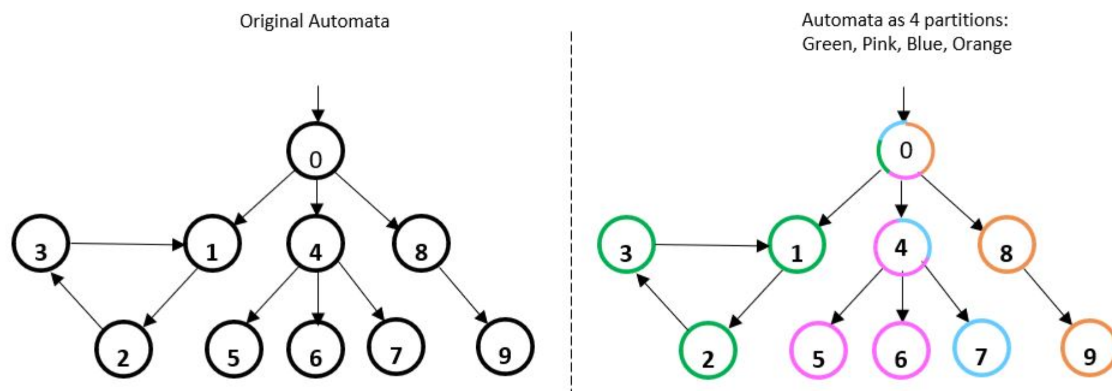



Figure 3.1 An example of partitioning NFA based on colors.

Although, NFA partitioning has significantly contributed in improving the performance in three types of dataset, namely Small NIDs, Bioinformatics, and Synthetic,



www.manaraa.com

the algorithm has not been tested on other types of automata applications such as Brill Tagging, Protomata and Random Forest, beside the algorithm has several restrictions. First, the algorithm manages the NFA as a tree, where only one node is the root node (or entry state). Second, any cycle is addressed as one super state that cannot be split into multiple partitions.

The second approach of NFA transformation is Fan-in Relaxation, which is proposed by Center of Automata Processing (CAP) at University of Virginia [36]. This approach differs than the above by managing the NFA as a graph, where more than one vertex can be entry (start) vertex and number of connected components can be found. This approach is used to transform the NFA which arbitrarily has large fan-in, and violates the hardware fan-in restrictions.

## 3.5 Comparative Studies of NFA Implementations on CPUs, GPUs, and FPGAs

Once configured with an NFA description, the Micron Automata Processor, the Unified Automata Processor, and all FPGA-based automata processors generally achieve high traversal throughput of one or two input symbols per clock cycle. Processing NFAs that are too large to fit on a device requires multiple passes of the input stream. Preprocessing time, which potentially includes synthesis and place-and-route, is often an important performance consideration. CPU- and GPU-based approaches can process NFAs stored in DRAM and are generally less affected by preprocessing time, but their traversal time–especially for larger NFAs–is limited by their cache performance. Since the behavior of automata processors is dependent on both the NFA structure and input stream, performance comparisons between competing architectures is difficult.

NFA descriptions such as ANML, NFA, or regular expressions are implemented for special-purpose automata representation. Such ANML description for Micron



المنارة للاستشارات
www.manaraa.com

AP, NFA for CPU and FPGA, and INFANT for GPU [17]. To simplify the comparisons between the three different platforms, [13] proposed MNCart as comprehensive central ecosystem gathering automata tools. MNCart system is represented by JSON-BASED open-source network language MNRL for representing the state machines.



المنارة للاستشارات
www.manaraa.com

# Chapter 4

# NAPOLY Design

In this chapter, we provide an overview of the architecture of our reusable Non-determinstic Automata Processor OverLaY (NAPOLY), its State Transition Elements, programmable interconnects, and its resource constraints. Then, we provide an overview about the I/O interface including the design of NAPOLY report region. Finally, we discuss NAPOLY performance model.

## 4.1 Overlay Architecture

NAPOLY architecture is reusable (without FPGA reconfigurations) across different NFA descriptions having arbitrary state labels and arbitrary logical NFA typologies, provided that the logical topology does not violate resource constraints inherent in the overlay structure. The most important constraint is a parameter of the interconnect that we refer to as "hardware fan-out", which determines the maximum number of outgoing transitions per STE as well as the maximum distance between a pair of connected STEs with respect to their location in the array. For example, with a hardware fan-out of 10, $STE_n$ can only connect to $STE_{n-4}$ to $STE_{n+5}$ (including itself). We developed several Pareto optimal versions of the overlay with varying numbers of STEs and hardware fan-out.



www.manaraa.com

#### 4.1.1 STE Design

Without considering the resource usage of the interconnect, the number of STEs is limited by the on-chip RAM available to store the input symbols associated with each STE.

Figure 4.1 shows the design of our STE. In order to achieve maximum utilization of the MLAB memory, the current state tables are generated as 256-deep x M bit RAMs, where M = the number of STEs. Each STE accepts a one-bit input from its corresponding column in the current state table, indexed by the input symbol.

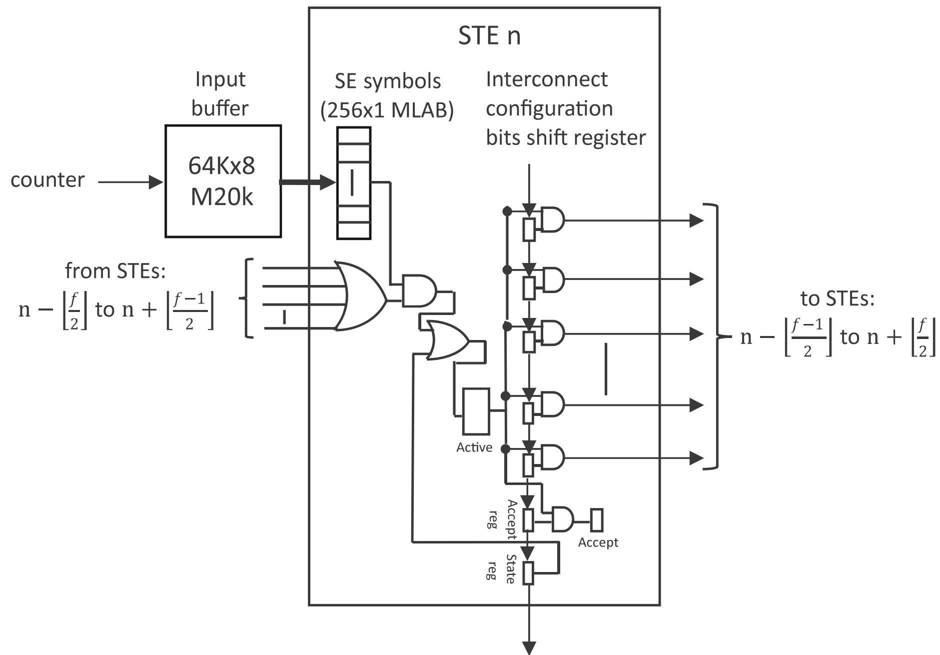



Figure 4.1 State Element STE Design

Each STE contains an OR-gate accepting activation signals from its connected predecessor STEs. Any cycle in which any of the incoming activation signals are asserted while receiving a one-bit from the current state table will activate the STE's state bit in the following cycle. Unless the start bit is set, the state bit resets in any cycle in which this condition does not hold. While the state bit is set, the STE will



المنارة للاستشارات
www.manaraa.com

broadcast an activation signal to all its outputs, which are each AND-gated against a corresponding interconnect configuration bit before being sent out to its successor STEs. The interconnect configuration bits and the start and reporting flags are stored in a set of flip-flops connected in a shift register both internally and across all the STEs in the array. As such, the number of available ALMs defines an upper bound on the level of interconnectivity.

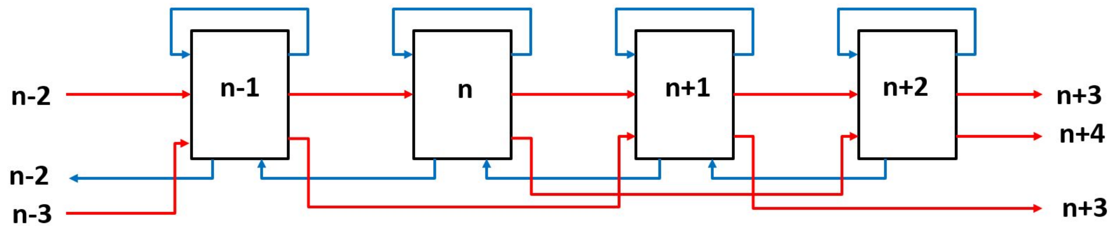



Figure 4.2 An example of NAPOLY interconnects.

#### 4.1.2 Interconnection Design

The physical STEs on the FPGA are connected using point-to-point links, where each STE sends an output signal to itself and $f-1$ of its neighbors, where f = the hardware fan-out. The STEs adopt a one-dimensional addressing scheme, where each STE has an ID number assigned contiguously and sends output signals to STEs $n - \lfloor\frac{(f-1)}{2}\rfloor$ to $n + \lfloor\frac{f}{2}\rfloor$, where n=the STE ID. Figure 4.2 shows NAPOLY interconnects when $n = 4$, and $f = 4$. The blue and red wires represent the backward and forward interconnects respectively.

This interconnect design is different than some previous ASIC designs, which use the hierarchical switched interconnect that gives each State Transition Element the ability to send signals to a larger pool of potential successor STEs. However, a switched interconnect complicates NFA preprocessing, as the synthesis tools must place and route the states onto the fabric while managing shared interconnect resources. On the other hand, our design requires only consideration of state-to-STE



المنارة للاستشارات
www.manaraa.com

mapping, since there are dedicated, non-shared wires between each pair of connectable STEs.

### 4.1.3 Overlay Resource constraints

STE capacity is bounded by RAM capacity. Our evaluation FPGA is an Intel Stratix 5 GX A7. This device has roughly 7X the on-chip memory capacity in M20K resources than it does in its MLAB (LUT-based) resources, but there are several practical problems with using M20K resources for the current-state tables. First, the M20K blocks are available in only 20 out of the 209 columns on the FPGA while the MLAB blocks are more uniformly distributed. Using MLABs avoids congestion around the M20K columns. Second, the current state tables have a depth of 256, while the minimum depth required to fully utilize M20K resources is 512, meaning that only 0.5 of the M20K capacity is available for depth-256 tables. Third, the M20K requires synchronous reads, which if used for the current state table would potentially reduce the throughput by 1/2, as each input symbol would require one cycle to access the current state table and another for updating the state flip-flop. Finally, the M20K blocks are needed for other purposes, such as to buffer the input and output data for the AP fabric. The Stratix 5 GX 7A contains 7.16 Mb of MLAB memory, giving an upper bound of roughly 29K STEs, as compared to 48K STEs on the Micron AP.

## 4.2 I/O Interface

The input symbols coming from the DRAM through the interface buses are stored into input buffer. The outputs reported in NAPOLY are stored into number of output buffers before flushing out to the DRAM.



المنارة للاستشارات
www.manaraa.com

#### 4.2.1 Input Buffer

A 64K x 8-bit M20K-based RAM serveing as the input buffer. Once filled, it streams input data into the STE array at one symbol per cycle (152 MB/s for the 4K-STE overlay). Filling the input buffer from DRAM requires $8.6\mu s$ ( 7.1 GB/s), performed $S/64K$ times, where $S$ is the total number of input characters.

#### 4.2.2 Output Buffer and Report Region

Any STE may be mapped to a particular reporting state, which causes it to generate a global output signal or “report” in all cycles in which it is active. Ideally the output buffer would accommodate a scenario where all states are configured as accepting states and all states are active in every cycle (this is easily achievable by setting the “start” and “reporting” flag on all STEs).

In order to obtain the reporting ID, NAPOLY is implemented to have multiple output regions, where each region represents a group of consecutive STEs ($M$). The number of reporting regions in the design is equal to ($\frac{N}{M}$), where N is total number of STEs. To determine which STE is reporting in each group, we used a priority encoder. The number of encoders determines the maximum number of reports per clock cycle without stalling.

#### 4.2.3 output buffer implementation

Unlike the input buffer which is 64KB x 8-bit, the output buffer is designed to have various depth and width depending on the overlay size and total number of priority encoders in each report region. The buffer depth depends on overlay size, smaller overlays have higher output buffer depth. The output buffer depth for overlay 4K, 8K, 12K, 16K, 20K and 24K respectively is 64K, 32K, 24K, 16K, 12K and 8K. To illustrate, let us assume the output region represents 1K STEs, and number of encoders per region is 4, and overlay size 8K STEs, this gives total number of regions is



المنارة للاستشارات
www.manaraa.com

8 ($= \frac{8K}{1K}$), and total number of encoders is 32 ($= 4 \times 8$). The output buffer comprises of output the buffer width of the 8K STEs and 32 encoders in total is equal to 320. This is considered the width of input port (port A) of the RAM. The output port (port B) is designed to allow the output buffer to access the DDR3 and transfer data using DMA. Therefore, its width is fixed to 512-bit, which equals to DMA signal width.

However, the dual RAM design is restricted by the set of ratios between port A and port B width are 1, 2, 4, 8, 16, 32. This prevents generating RAM with ratio $\frac{WidthB}{WidthA}$, leading us to necessitate padding the input port width by extra 0s to the left in order to achieve the minimum valid ratio between the two ports. These padded bits are used to store the input offset in as shown in Figure 4.4.

#### 4.2.4 Priority Encoder Operation

The priority encoders in each group works simultaneously to identify the active report STEs per time. The process starts by the right-most bit in the group, checking if the bit is set. If so, the bit will be encoded and its ID sent to the reporting-ID register. If the bit is zero, the priority encoder moves to the very next bit and repeat the process, until the final bit in the group.

Figure 4.4 shows the design of priority encoder in our report region. In the Figure, we assume total number of STEs (overlay size) is 16. The size of the output region is 8, so the number of groups is 2 ($= \frac{16}{8}$). We instance that the number of priority encoders in each group is 4. During the reporting time, in addition to store the ID of the reporting STE, the offset of the input symbol is also being stored in order to identify the match location in the input sequence. Both the STE IDs and offsets are stored into a M20K-based dual RAM serves as the output buffer.



www.manaraa.com

## 4.3 NAPOLY Performance Model

Historically, automata implementations have evaluated performance in terms of symbol throughput, such as symbol per cycle or symbol per second. However, this assumes that the entire input automata will fit on a single chip. Practical workloads inevitably require multiple passes, and reconfiguration time plays a substantial part of end-to-end performance. Estimating performance is mostly not only, a matter of estimating the number of reconfigurations, but also estimating the time to read input set and flush the reports.

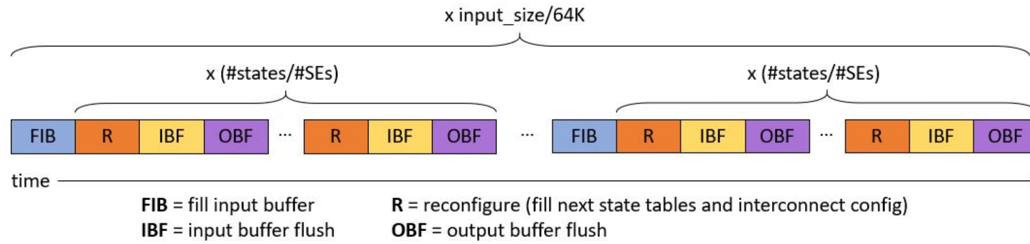



Figure 4.3 NAPOLY Timing Diagram

At runtime, NAPOLY follows the timing diagram shown in Figure 4.3. For each block of input characters the array must fill the input buffer from DRAM ($\frac{size_{input_buffer}}{bw_{DRAM}}$), and for each batch of STEs it must reconfigure its array ($time_{reconfig}$) (loading next_state tables and configuring gates), flush the input buffer through the array ($time_{IBF}$), and flush the output to DRAM ($time_{OBF}$). Time to configure gates depends on **f** and **STEs**, while time to load the next state tables depends on **STEs** multiplied by 256. Therefore, Time to load the next state table grows by the increase in overlay capacity, while time to load gates decays over the overlay capacity as the **f** decreases. For a given NFA and input, the effective throughput is calculated according to Equation 4.1.

The reconfiguration time $time_{reconfig}$ gives the time needed to reconfigure a new NFA onto the overlay. Thus the execution time scales with $R \times time_{reconfig} \times \frac{IS}{64K}$,



www.manaraa.com

where 64 KB = the size of the input buffer and $IS$ is the size of the input data to be searched for patterns.

$$\text{Throughput} = \frac{size_{input_buffer}}{\frac{size_{input_buffer}}{bw_{DRAM}} + R \times (time_{reconfig} + time_{OBF} + time_{IBF})} \tag{4.1}$$



المنارة للاستشارات
www.manaraa.com

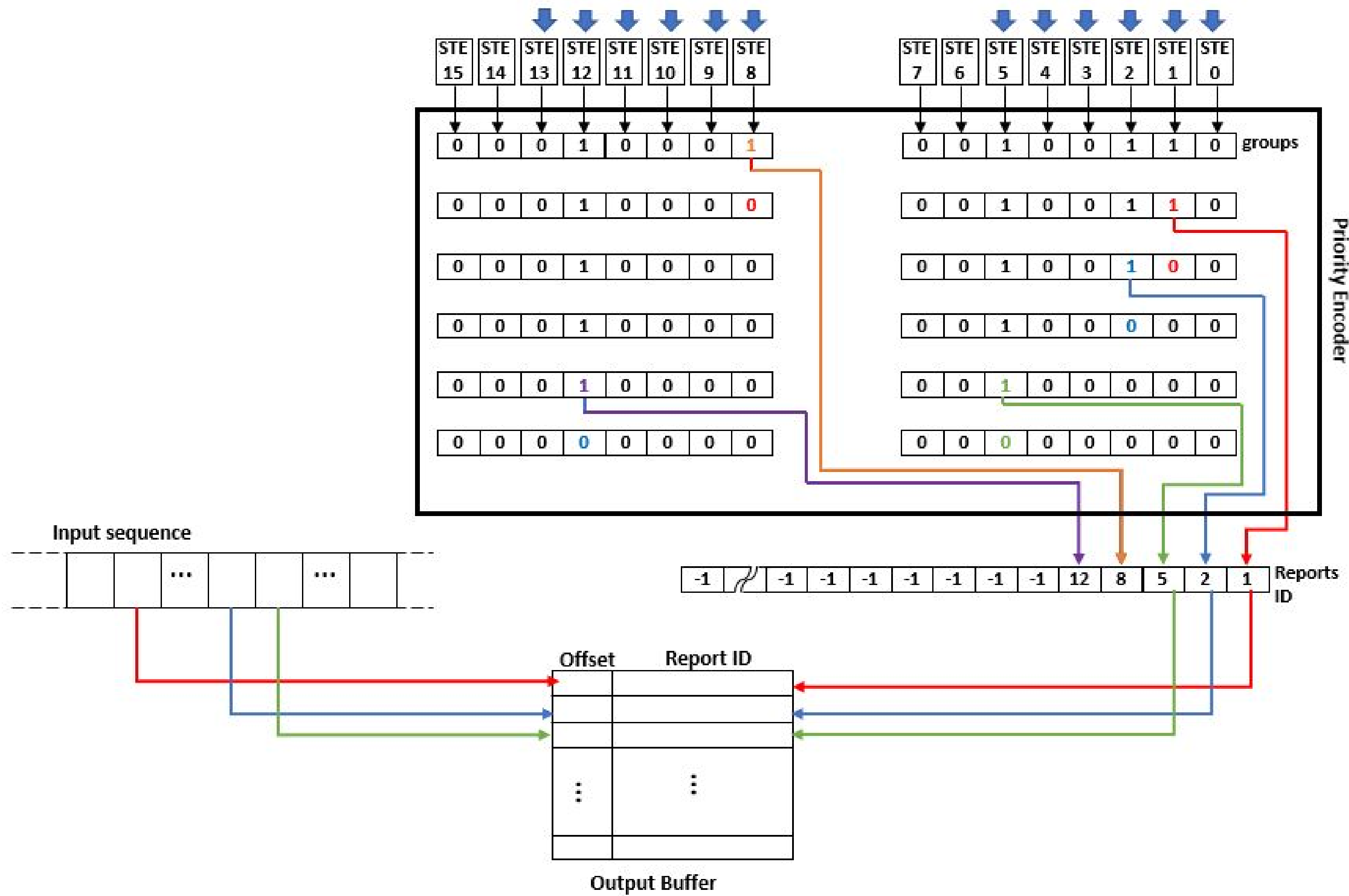



Figure 4.4 NAPOLY Output Region.



المنارة للاستشارات
www.manaraa.com

# Chapter 5

# Mapping Problem

Mapping an NFA graph to an overlay (NAPOLY) is performed by allocating every state into STE and mapping every edge to physical interconnect without violating the hardware fan-out constraints.

**Definition 2.** For a given NFA $\{V, E\}$, where $V$ is a set of states and $E$ a set of edges (transitions), a **map** is an association between each of the NFA states of an NFA graph and a corresponding STE index in the range of $[0, N-1]$, where $N =$ number of STEs. There are thus $|V|!$ unique maps for a given NFA assuming $|V| = N$.

For example, assume we have NFA graph consisting of 7 states [A, B, C, D, E, F, G] as shown in Figure 5.1, and we wanted to map this NFA onto an overlay consisting of 7 STEs [0, 1, 2, 3, 4, 5, 6]. Assume the hardware fan-out f is 9, forward connections are 4, backward connections are 4, and self loop is one connection.

If we map each state to a STE in order, as it is shown in Figure 5.1, the edge between B and G will require a connection to mapping state B onto $STE_1$, and mapping state G to $STE_6$. This will violate the fan-out constraint which is maximum "reach" of 4.

One way to pass the mapping is to map state F to $STE_6$, and G to $STE_5$, as shown in the Figure. There is N! possible ways to map NFA graph, however not all the ways can achieve the mapping successfully. The number of successful mapping solutions increases by growing the hardware fan-outs.



المنارة للاستشارات
www.manaraa.com

In this particular example, there is 7! or 5040 possible ways to map such small NFA. With a hardware fan-out of 5, there is no mapping solution. By increasing hardware fan0out, the number of possible solutions multiplies. As instance, with a hardware fan-out of 6, 7, and 8, the number of mapping solutions significantly grows to 24, 48, and 372 respectively.

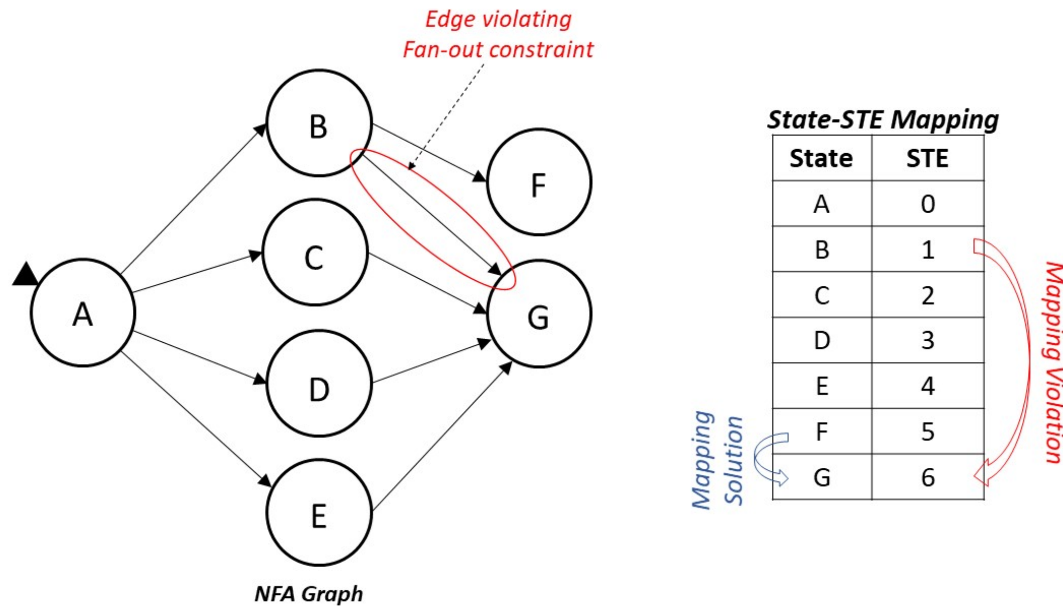



Figure 5.1 Mapping Problem.

For the purpose of achieving effective mapping solutions, two approaches are presented in this chapter: Heuristic and SAT solver. In this chapter, we also presents the NFA transformation and its impacts on overall performance.

## 5.1 Greedy Mapping Heuristic

To arbitrary map NFAs into a target overlay having given a hardware fan-out, We have developed a greedy mapping heuristic.

The hardware fan-out determines the number of wire tracks to and from each STE, as well as the maximum reach of each STE in terms to maximum distance over



المنارة للاستشارات
www.manaraa.com

which a connection can be made between two STEs: $i - j \leq \lfloor \frac{f-1}{2} \rfloor$ and $j - i \leq \lfloor \frac{f}{2} \rfloor$ for hardware fan-out $f$, for any edge in the NFA description $s \rightarrow d$ where state $s$ is mapped to STE $i$ and state $d$ is mapped to STE $j$.

The hardware fan-out parameter is a constraint that defines which subset of maps are valid for a given NFA. In order to find a valid map, a mapping algorithm must solve the following problem.

Given a set of NFA edges $\{e : \forall(p, s) \in E\}$ , find:

$$\left\{\begin{array}{lr} map(p), map(s) : & \\ (map(p), map(s) \text{ are unique}) & \text{and} \\ (map(p) - map(s) \leq \lfloor (f-1)/2 \rfloor) & \text{and} \\ (map(s) - map(p) \leq \lfloor f/2 \rfloor) & \end{array}\right\}$$

**Definition 3.** For a given NFA graph $\{V, E\}$ and a given map, a **mapping violation** is any edge $(p, s) \in E$ where $(map(p) - map(s) > \lfloor (f-1)/2 \rfloor)$ or $(map(s) - map(p) > \lfloor f/2 \rfloor)$. In other words, a mapping violation occurs for each NFA edge whose predecessor and successor states are mapped to STEs whose indices are too far apart given the hardware fan-out of the target NAPOLY interconnect.

For a given map, our heuristic greedily finds and resolves each mapping violation. Our heuristic resolves each violation in order of ascending predecessor STE index by reallocating the offsets, meanign all the mappings between the new location and the original locations, which has unpredictable offsets on the score.

The score function is computed as shown in Equation 5.1.

$$\sum_{(p,s) \in E} |\text{map}(p) - \text{map}(s)| \tag{5.1}$$

The score function is the accumulated mapped distance of the mappings of each predecessor-successor pair, where the distance is defined as the difference in STE index. The score is not directly affected by mapping violations, meaning that mapping



المنارة للاستشارات
www.manaraa.com

$A$ could have a lower score than mapping $B$ when mapping $A$ has more violations than mapping $B$.

We found that this approach gives the mapper flexibility to make decisions that potentially increase the number of mapping violations in order to achieve longer-term optimization. A consequence is that violations are likely to still exist after each pass through the STEs, in which case the heuristic will make additional passes as needed to resolve all violations. The mapping heuristic is explained in details in Appendix A.

### 5.2 SAT solver mapping algorithm

Mapping states to STEs against the hardware interconnect constraints is an NP-complete, reducible to SAT problem. The hardware fan-out parameter defines which subset of maps are valid for a given NFA. In order to find a valid map, a mapping algorithm must assign $map(s) \forall s \in V$ subject to the following constraints:

1. **Maximum hardware fan-out**,

   $\forall (s,d) \in E$: $((map(s) - map(d)) \leq \lfloor(\frac{f-1}{2})\rfloor)$ *and* $((map(d) - map(s)) < \lfloor(\frac{f}{2})\rfloor)$

2. **Every state must be assigned to only one STE**

   $\forall s \in V, \forall i,j \in N, i \neq j,$ *if* $map(s) = i,$ *then* $map(s) \neq j$

3. **Every STE must be allocated one state**

   $\forall s,d \in V, s \neq d, \forall i \in N,$ *if* $i = map(s),$ *then* $i \neq map(d)$

4. **All states must be allocated**

   $\forall s \in V, map(s) \in N$

In order to allocate the states into STEs, we describe the constraints above in conjunctive normal form (CNF), which is a conjunction of clauses, where each clause is formed as a disjunction of literals.



المنارة للاستشارات
www.manaraa.com

We assign each possible mapping of a state to an STE as a Boolean variable whose state determines if the mapping is made, i.e. Let $L_i^s = TRUE$ when $map(s) = i$.

We describe *constraint 1* as shown in Equation 5.2 by constructing a set of clauses that collectively guard against every possible mapping violation defined in 3:

$$\bigwedge_{\forall (s,d) \in E, \forall i \in N} (\overline{L_i^s} \wedge \bigvee_{\forall m \in [-\lfloor \frac{f-1}{2} \rfloor ..., -1, 1, \lfloor \frac{f}{2} \rfloor]} L_{i+m}^d) \tag{5.2}$$

In other words, if any edge $(s, d)$ is mapped such that $map(s) == i$ and state $d$ is not mapped to SEs $i - \lfloor \frac{f-1}{2} \rfloor$ to $i + \lfloor \frac{f}{2} \rfloor$, then the clause will be FALSE, invalidating the entire CNF expression.

We describe *constraint 2* by adding an additional clause for each state, comprised of the conjunction of the literals representing every possible mapping of that state, as shown in Equation 5.3.

$$\bigwedge_{\forall s \in V} \bigvee_{foralli \in N} L_i^s \tag{5.3}$$

We describe *constraint 3* by adding $|V|^2 \times |N|$ additional clauses, formed from the conjunction of the complimented variables corresponding to every pair of states mapped to every STE, as shown in Equation 5.4.

$$\bigwedge_{\forall s_1 \in V} \bigwedge_{\forall s_2 \in V} \bigwedge_{\forall i \in N} (\overline{L_i^{s_1}} \vee \overline{L_i^{s_2}}) \tag{5.4}$$

We describe *constraint 4* similar to the previous constraint, but for each conjunction as the complimented variables corresponding to each state mapped to every pair of STEs, as shown in Equation 5.5.

$$\bigwedge_{\forall i_1 \in N} \bigwedge_{\forall i_2 \in N} \bigwedge_{\forall s \in V} (\overline{L_{i_1}^s} \vee \overline{L_{i_2}^s}) \tag{5.5}$$



المنارة للاستشارات
www.manaraa.com

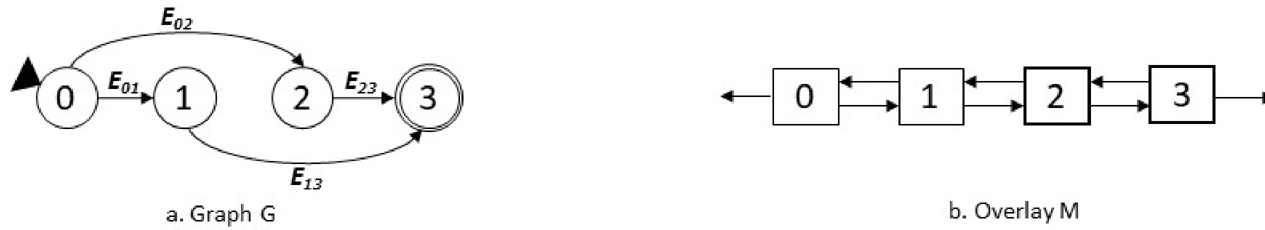
Figure 5.2 depicts an example NFA, overlay, and corresponding CNF clauses that describe constraint 1. Graph $G$ is composed of $V \in 0, 1, 2, 3$, $E \in (0, 1)$, $(0, 2)$, $(1, 3)$, $(2, 3)$, and overlay $M$ is composed of $N \in (0, 1, 2, 3)$ and $f = 3$.

Each potential mapping clause is shown as a matrix in Figure 5.2 where its rows represent the state end the column represent the STEs to which the state can potentially by mapped. The cells in the matrix are the literals of the clauses, shown as T for the positive literal and F as the negative literal. The clause joins literals by OR, while clauses are joined by AND.

The example presents two cases: (1) mapping $state_0$ to all possible STEs and mapping its successors into STEs based on $state_0$ allocation and the physical distance or the hardware fan-out. If assigning $state_0$ to any STE is false, its successor state 1 must be true into either $STE_1$ or $STE_2$. Same for case (2) when mapping $state_2$ and its successors into all possible STEs.

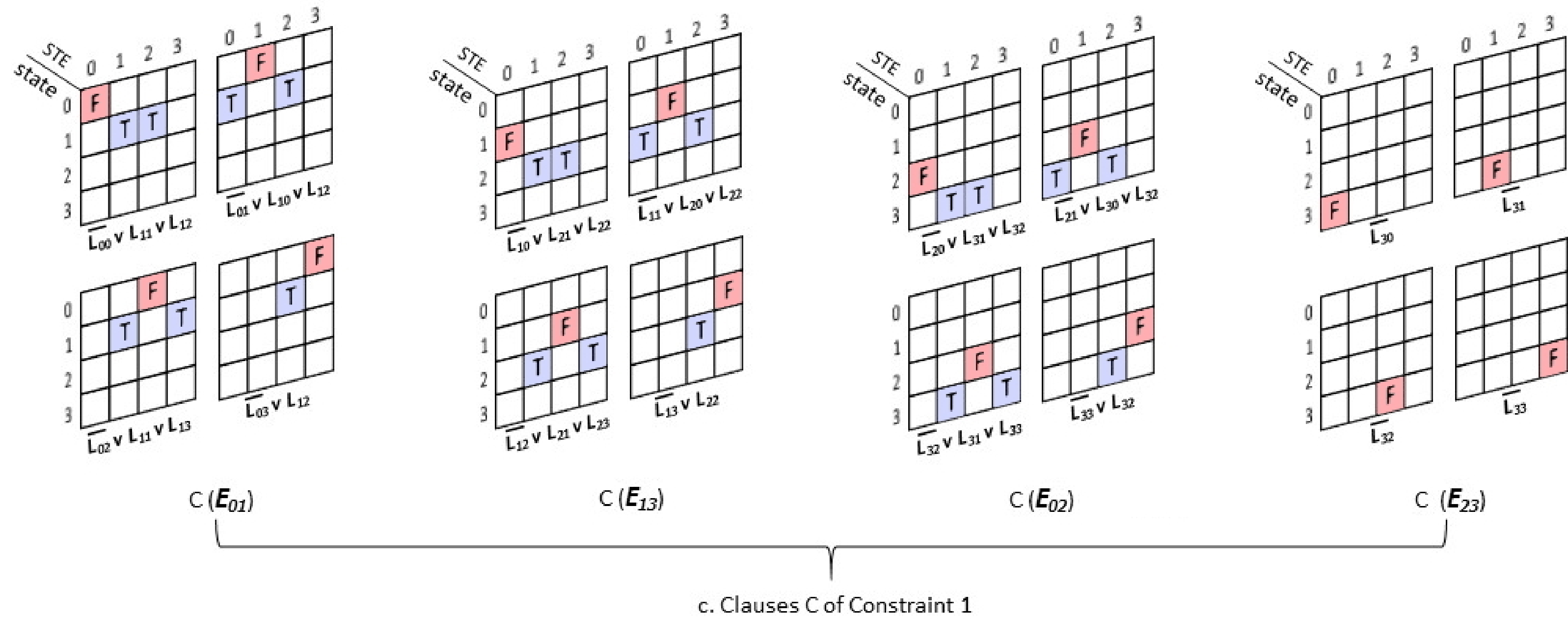


المنارة للاستشارات
www.manaraa.com




a. Graph G

b. Overlay M




c. Clauses C of Constraint 1

Figure 5.2 An example for generating CNF clauses of literals based on *Constraint 1*.



المنارة للاستشارات
www.manaraa.com

## 5.3 NFA Transformation

When synthesizing a NAPOLY overlay configuration, there is a tradeoff between the number of STEs and the hardware fan-out. As shown in Figure 6.2, larger overlays achieve higher throughput because they require less runtime reconfigurations. For a given benchmark automata, the maximum overlay size available to it is limited by the minimum hardware fanout on which it can be mapped. As such, our goal is to map every NFA onto an overlay having minimal hardware fan-out, since this will allow for the use of a larger overlay. The minimum hardware fanout depends on the density of the automata, which can characterized by its logical fan-in and fan-out (the maximum number of incoming and outgoing transitions, respectively). In general, these values have an affect on the minimum hardware fanout for which our mapper can map the automata.

Our methodology for finding the minimal hardware fan-out for a given NFA is to perform a trial-and-error binary search. For some benchmarks, it is possible to reduce the hardware fan-out by transforming the NFA into a functionally equivalent alternative form that limits the maximum number of incoming and/or outgoing transitions from each state. In some cases, this allows for the use of an overlay with more STEs and less hardware fan-out than would otherwise be required at the cost of an increased number of states.

Specifically, we use the fan-in/fan-out relaxation technique included in VASim [37] to decompose any subgraph that has any states that have an in- or out-degree larger than the prescribed fan-in or fan-out limit into two or more smaller graphs. This type of transformation replicates all the states along all the paths from the start states to the accepting states that are part of any of the high fan-in or fan-out paths, as shown in the example in Figure 5.3. As such, this approach is only practical when the performance gained from increasing the overlay size outweighs the performance loss caused by increasing the number of states and resulting number of reconfigurations.



المنارة للاستشارات
www.manaraa.com

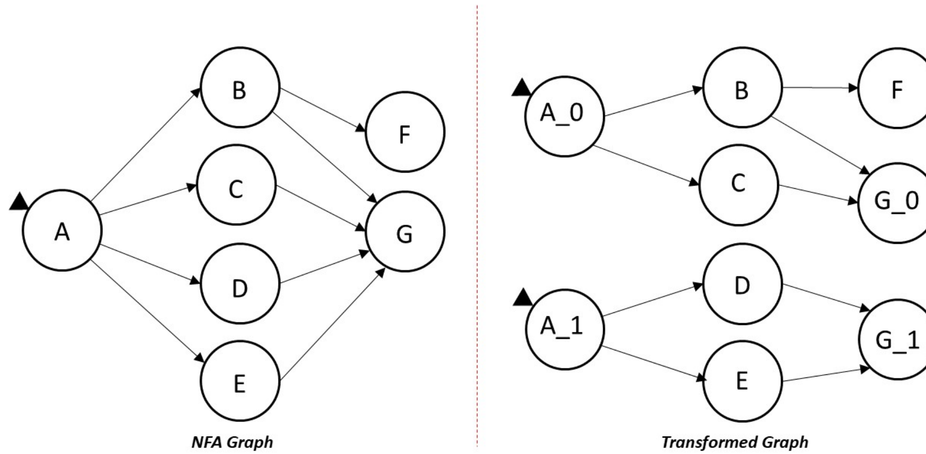
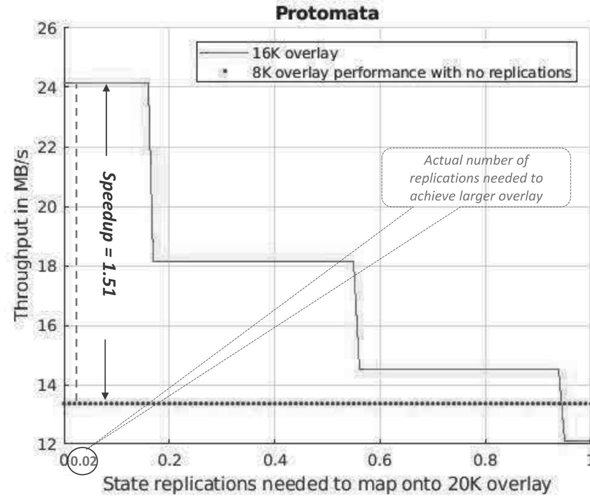



Figure 5.3 Transformation of NFA graph in Figure 5.1.

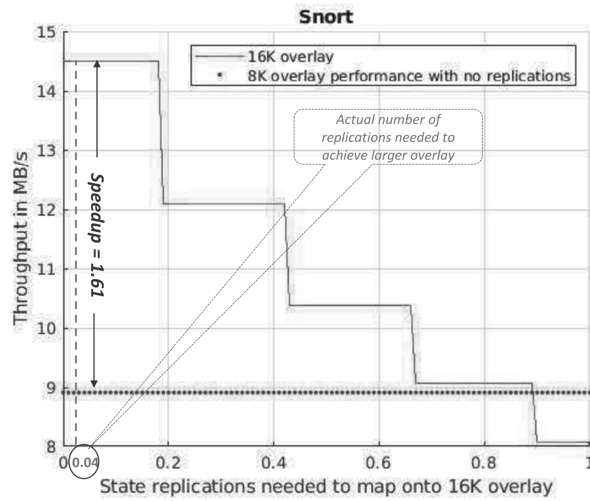
We applied this technique to each NFA benchmark in which less than 5% of its NFA sub-graphs failed to map with the hardware fan-out needed to migrate the benchmark to the next larger overlay. Figure 5.4 shows the potential speed up versus number of state replications needed to enable upgrading the overlay in 3 of the benchmarks. Figure 5.4 shows Promomata, Snort and PowerEn performance when tolerating number of states to improve the performance. Increasing the number of states by 0.02 speeds Protomata performance up by 1.51, as shown in (a). As shown in (b), the actual number of state replications 0.04 is needed to obtain the best performance of Snort at 16K overlay. Power En throughput, in Figure (c), can be improved by replicating the states by 0.008 and upgrading the overlay to 20K. As shown Power En performance speeds up only within very small region (number of replications $\leq 0.003$) of 20K overlay performance plot.



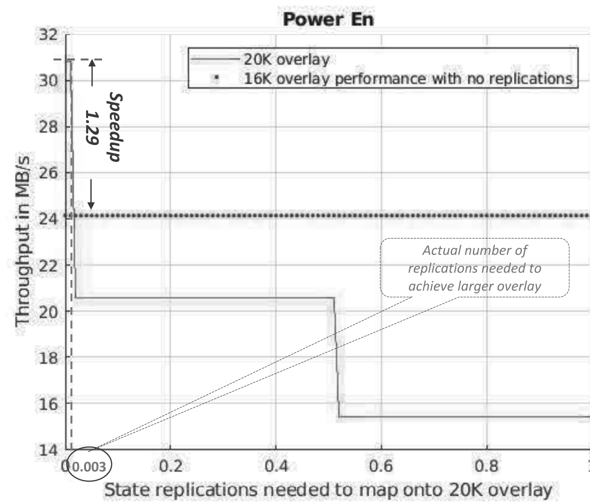
المنارة للاستشارات
www.manaraa.com




(a) Protomata




(b) Snort




(c) Power En

Figure 5.4 Speeding up performance versus replications.



www.manaraa.com

# Chapter 6

# Experimental Analysis

In this chapter, our experimental analyses and results are presented in five main sections: Hardware Resources, Mapping automata into the overlay, the improvement achieved in performance after transforming the NFA, comparing NAPOLY performance with state-of-the-art GPU and CPU, and performance with scalability.

## 6.1 Hardware Resources

Table 6.1 shows the hardware resources consumed to implement 6 different overlay configurations. The column labeled **#STEs** gives overlay sizes (number of STEs) and the column labeled **Maximum hardware fan-out** gives **f** the max reach of every STE. As shown in the Table, there is a tradeoff between the STEs and the achieved fan-out. Also, larger overlays achieves lower maximum frequency (Fmax) in MHz, as shown in the column labeld **Fmax**.

Table 6.1 Hardware Resources Used in Different Overlay Configurations

| # STEs | Max Hw. Fan-out (f) | Fmax (MHz) | MLABs | ALMs% | Reg.% | M20K% |
|---|---|---|---|---|---|---|
| 4K | 103 | 152 | 1,047,296 | 90 | 46 | 41 |
| 8K | 44 | 136 | 2,096,384 | 91 | 41 | 41 |
| 12K | 25 | 122 | 3,145,472 | 95 | 36 | 60 |
| 16K | 12 | 121 | 4,193,024 | 94 | 26 | 41 |
| 20K | 6 | 119 | 5,242,112 | 95 | 19 | 61 |
| 24K | 3 | 112 | 6,291,200 | 96 | 15 | 41 |



المنارة للاستشارات
www.manaraa.com

Table 6.2 Total M20K Used for Output Buffer

| # STEs | Buffer Depth | Buffer Width | Buffer Width After Padding | Total M20K (MB) |
|---|---|---|---|---|
| 4K | 64 | 192 | 256 | 16 |
| 8K | 32 | 416 | 512 | 16 |
| 12K | 24 | 624 | 1024 | 24 |
| 16K | 16 | 896 | 1024 | 16 |
| 20K | 12 | 1144 | 2048 | 24 |
| 24K | 8 | 1399 | 2048 | 16 |

The column labeled **MLABs** gives the total number of MLABs used in every overlay. This number increases at larger overlays since the MLAB is a simple dual-port SRAM used to implement the next-state table in each STE. The number of MLABs is theoretically equivalent to $\#MLABs = \#STEs \times 256$, and each next-state table is $256 \times 1$ RAM. As shown in the table, **ALMs** usage is almost the same in all the overlays, while percentage of **Regs**, which limits the maximum hardware fan-out, lowers with larger overlays. Finally, the column labeled as **M20K** shows the percentage of M20K, which is used in implementing the input and output buffers. The input buffer is $64K \times 8 - bit$ in all the overlays. The output buffer size depends on the overlay.

Table 6.2 shows the total M20K used to implement the output buffer in the overlays. The column labeled **Buffer depth** ranges between $[64K, 32K, 24K, 16K, 12K, 8K]$ based on overlay size (**#STEs**). The Column **Buffer Width** shows the width of the output buffer, which is determined by $\#\text{encoders} \times \#\text{outputregions} \times \log_2(\text{STEs})$. As described in Chapter 5, the buffer width needs to be padded, as shown in column **Buffer Width after Padding**. Column **Total M20K** shows the total number of M20K needed in each overlay.



المنارة للاستشارات
www.manaraa.com

Table 6.3 Repertoire of the achieved NAPOLY Configurations

| # STEs | Max BW (GB/s) | Time Reconfig T ($\mu$s) | Outp Encod. | Max Report Cycles | Max Report Rate (GHz) | Throughp 24K states (MB/s) | Throughp 128K states (MB/s) |
|---|---|---|---|---|---|---|---|
| 4K | 1866 | 21 | 16 | 100% | 2.4 | 14 | 3 |
| 8K | 1427 | 31 | 32 | 50% | 2.2 | 27 | 5 |
| 12K | 1031 | 43 | 48 | 33% | 2.0 | 32 | 6 |
| 16K | 692 | 53 | 64 | 25% | 1.9 | 36 | 9 |
| 20K | 426 | 67 | 80 | 20% | 1.9 | 31 | 9 |
| 24K | 240 | 74 | 96 | 17% | 1.8 | 67 | 11 |

## 6.2 NAPOLY Run Time

Tables 6.3 shows the Pareto optimal set of synthesized and place-and-routed overlay configurations with respect to STE capacity and hardware fanout. The first column of the table **#STEs** lists all the 6 NAPOLY configurations. The column labeled **Max BW for** $N_{\%active} = 0.25(GB/s)$ gives the upper bound for on-chip memory bandwidth needed for 25% active states. Exploitation of on-chip memory bandwidth is the principle performance advantage of NAPOLY over CPU- and GPU-based approaches. The column labeled **Time_Reconfig (T)** lists the time needed to reconfigure a new NFA onto the overlay. The column labeled **Output Encoders** gives the number of output encoders, which determines the maximum number of “reports”, or accepting state activations, allowed per clock cycle.

Likewise, the column labeled **Max Reporting Cycles** gives the depth of the output buffer relative to the depth of the input buffer (64K). Together, these values and Fmax determine the maximum reporting rate of the overlay configuration, listed in the column labeled **Max Report Rate (GHz)**. For a given NFA and input, the effective throughput is calculated according to Equation 4.1, which is shown in the last two columns (**Throughput for 24K** and **Throughput for 128K**) at 24K states and 128K states respectively. Figure 6.1 shows NAPOLY execution time is dominated by the time to flush input buffer and the time to flush the output buffer.



المنارة للاستشارات
www.manaraa.com

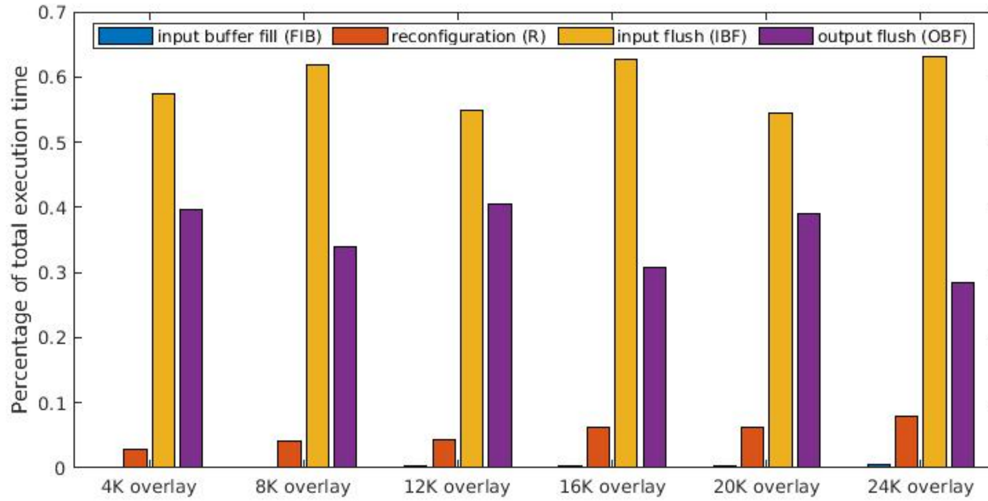



Figure 6.1 Execution time makeup of NAPOLY.

Figure 6.2 plots the throughput of all NAPOLY overlays for 1 million input characters and for a total NFA workload from 4K to 128K states. Overlays with higher STE capacity perform better for larger NFAs, but for greater than 100,000 states the performance differential is only 10%, indicating that the choice of overlay configuration has an increasingly small impact for increasingly larger NFAs.

## 6.3 Mapping Results

To evaluate the suitability of the mapping heuristic for realistic workloads, we mapped each of the NFA benchmarks in the ANMLZoo benchmark suite [11], which consists of 12 benchmarks from various applications as shown in Table 6.4, where the first column **Benchmarks** lists the benchmarks and second column **#States** shows total number of states in each benchmark. The key goal of this work is to find the minimal hardware fan-out under which we can map each benchmark.

Initially, we used the mapping heuristic to map ANMLZoo benchmarks. The mapping heuristic used will run infinitely when it cannot find a valid mapping, so it will abort execution when the derivative of the mapping score remains zero after



المنارة للاستشارات
www.manaraa.com

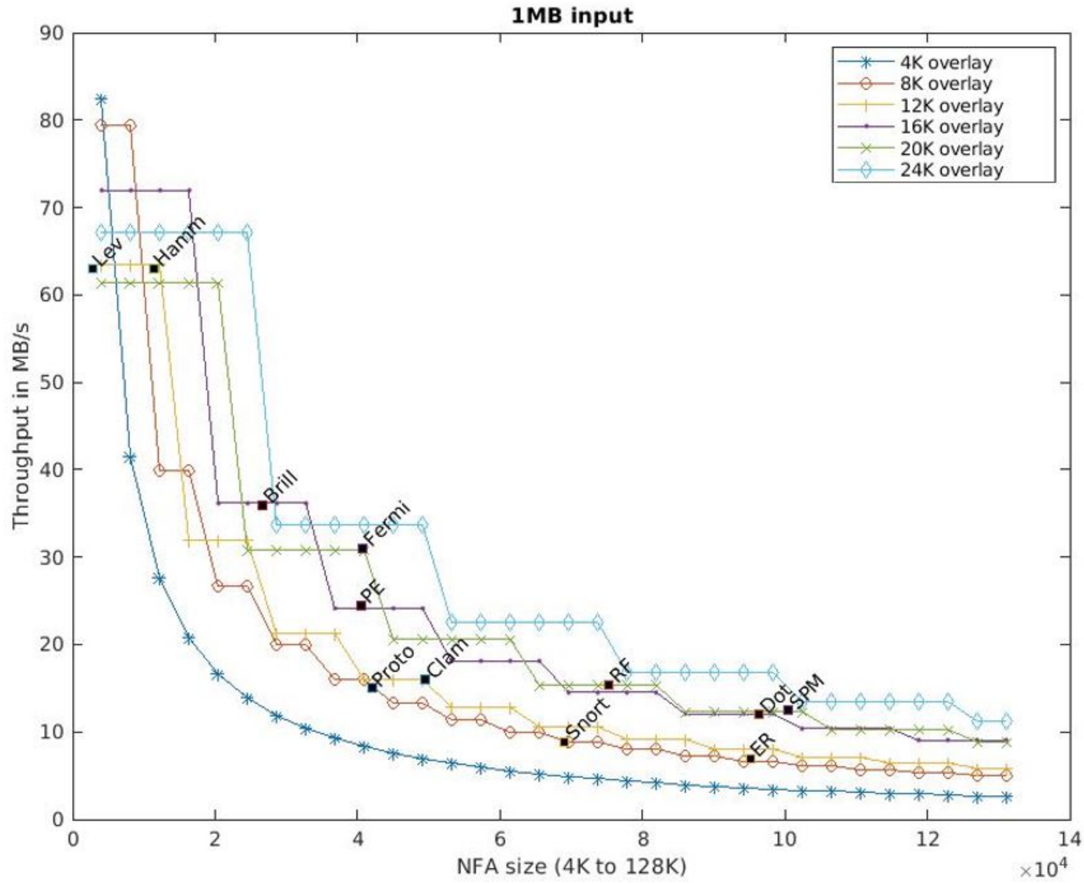



Figure 6.2 NAPOLY Performance vs. NFA size.

several iterations of **validate_edges**, and try again with a larger hardware fan-out value. After finding a valid mapping, a suitable NAPOLY overlay is chosen based on the needed hardware fan-out. The overlay always has less STEs than states, but enough STEs to hold the largest distinct graph in the benchmark (all ANMLZoo benchmarks contain multiple distinct graphs).

Table 6.4 shows the mapping result for each of the ANMLZoo benchmarks. The **Minimum f Achieved** column lists the minimum hardware fan-out required for each benchmark based on our heuristic mapping algorithm. The **Overlay Size** shows the largest target overlay that can support the needed fan-out. NAPOLY **re-configurations** is computed $\lceil \frac{S}{N} \rceil$, as shown in the fifth column. The column la-



المنارة للاستشارات
www.manaraa.com

Table 6.4 NAPOLY Mapping Using Mapping Heuristic

| Benchmarks | # States (S) | Min f Achieved | Overlay Size (N) | # Reconf. | Throughput (MB/s) |
|---|---|---|---|---|---|
| Brill | 26668 | 40 | 8K | 4 | 20 |
| Clam AV | 49538 | 18 | 12K | 5 | 13 |
| Dot Star | 96438 | 4 | 20K | 5 | 12 |
| ER | 95136 | 62 | 4K | 19 | 3 |
| Fermi | 40783 | 8 | 16K | 2 | 24 |
| Hamming | 11346 | 21 | 12K | 1 | 63 |
| Levenshtein | 2784 | 17 | 12K | 1 | 63 |
| Power En | 40513 | 29 | 8K | 5 | 16 |
| Protomata | 42061 | 48 | 4K | 10 | 13 |
| Random Forest | 75340 | 12 | 16K | 5 | 15 |
| Snort | 69029 | 60 | 4K | 17 | 5 |
| SPM | 100500 | 8 | 16K | 5 | 10 |

beled **Reconfiguration Time Throughput** lists the effective throughput for each benchmark, which includes the target overlay's clock speed and reconfiguration time.

Table 6.5 shows the mapping result for each of the ANMLZoo benchmarks using SAT solver. The Table is structured similar to Table 6.4. Comparing with mapping heuristic, SAT solver achieved a significant improvement in hardware fan-out, targeting larger overlay and reducing the number of re-configurations in 75% of ANMLZoo benchmarks, and Figure 6.3 shows the performance speed up for these benchmarks.

## 6.4 NFA Transformation Results

We applied NFA transformation technique (explained in previous Chapter) on Protomata, Snort and Power En benchmarks, which have less than 5% of its NFAs failing to map with the hardware fan-out needed to migrate the benchmark into next larger overlay.



المنارة للاستشارات
www.manaraa.com

Table 6.5 NAPOLY Mapping Using SAT solver

| Benchmarks | # States (S) | Min f Achieved | Overlay Size (N) | # Reconf. | Throughput (MB/s) |
|---|---|---|---|---|---|
| Brill | 26668 | 8 | 16K | 2 | 36 |
| Clam AV | 49538 | 12 | 16K | 3 | 16 |
| Dot Star | 96438 | 4 | 20K | 5 | 12 |
| ER | 95136 | 41 | 8K | 12 | 7 |
| Fermi | 40783 | 5 | 20K | 2 | 31 |
| Hamming | 11346 | 14 | 12K | 1 | 63 |
| Levenshtein | 2784 | 16 | 12K | 1 | 63 |
| Power En | 40513 | 8 | 16K | 3 | 25 |
| Protomata | 42061 | 42 | 8K | 6 | 15 |
| Random Forest | 75340 | 6 | 20K | 4 | 16 |
| Snort | 69029 | 36 | 8K | 9 | 9 |
| SPM | 100500 | 6 | 20K | 5 | 13 |

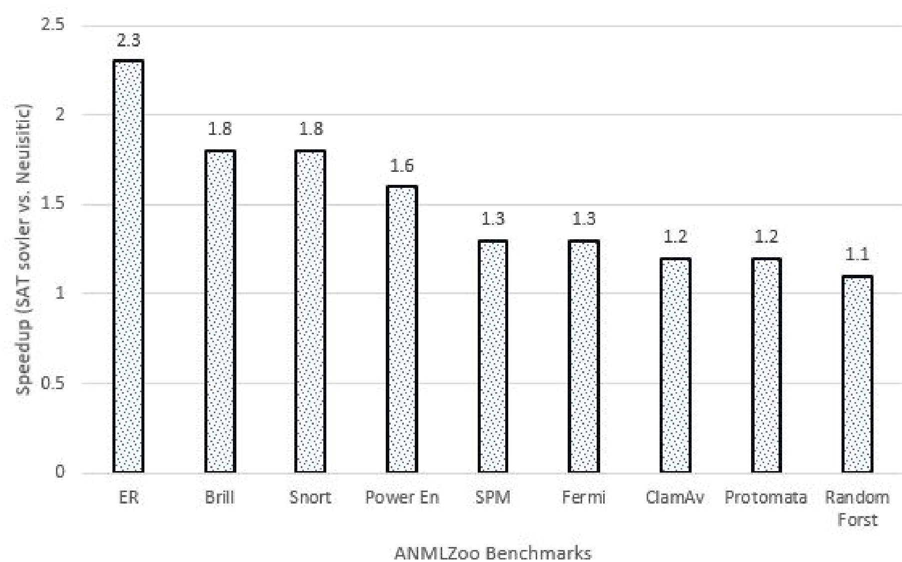



Figure 6.3 Speedup achieved in 75% of Benchmarks at SAT solver vs. heuristic.



المنارة للاستشارات
www.manaraa.com

Table 6.6 Snort Transformation Results

| Logical Fan-in/-out Limit | State Replications | Achieved HW Fan-out | Target Overlay |
|---|---|---|---|
| 10/10 | 0 | 36 | 8K |
| 8/8 | 1% | 12 | 16K |
| 6/6 | 3% | 11 | 16K |
| 4/4 | 4% | 11 | 16K |
| 2/2 | 4% | 9 | 16K |
| 1/1 | 40% | 2 | 24K |

Table 6.7 Protomata Transformation Results

| Logical Fan-in/-out Limit | State Replications | Achieved HW Fan-out | Target Overlay |
|---|---|---|---|
| 24/24 | 0 | 42 | 8K |
| 16/16 | 0.07% | 11 | 16K |
| 8/8 | 0.2% | 11 | 16K |
| 2/2 | 2% | 9 | 16K |
| 1/1 | 12415% | 2 | 24K |

Tables 6.7, 6.6, 6.8 show the state replications and the achieved hardware Fan-out after NFA transformation for three benchmarks Protomata, Snort and Power En. The first column represents the **Fan-in/Fan-out limit** applied on the failing sub-graphs of each benchmark. The second column, **State Replications**, shows the number of state replications achieved when fan-in/out limits applied. Third column shows the **Minimum Hardware Fan-out** achieved to map the sub-graphs onto larger overlays, and final column shows the **Target Overlay**. As shown in the three tables, the number of state replications significantly increases when limiting Fan-in/Fan-out to 1, while it lowers when moving the limits towards the maximum logical Fan-in/out for each benchmark.



المنارة للاستشارات
www.manaraa.com

Table 6.8 Power En Transformation Results

| Logical Fan-in/-out Limit | State Replications | Achieved HW Fan-out | Target Overlay |
|---|---|---|---|
| 4/3 | 0 | 8 | 16K |
| 2/2 | 0.3% | 6 | 20K |
| 1/1 | 10% | 2 | 24K |

Table 6.9 Performance Results

| Benchmark | NAPOLY Throughput | Average Active States (AS) | R per B | GPU Throughp (MB/s) | CPU Throughp (MB/s) | Speedup vs Max (CPU, GPU) |
|---|---|---|---|---|---|---|
| Brill | 36 | 14 | 4 | 7 | 1 | 9 |
| Clam AV | 16 | 4 | 5 | 4 | 14 | 1.14 |
| Dot Star | 12 | 3 | 5 | 40 | 10 | 0.3 |
| Entity Resolution | 7 | 10 | 19 | 4 | 1 | 1.75 |
| Fermi | 31 | 3854 | 2 | 2 | 1 | 15.5 |
| Hamming | 63 | 240 | 1 | 18 | 10 | 3.5 |
| Levenshtein | 63 | 88 | 1 | 38 | 1 | 1.65 |
| Power En | 31 | 31 | 5 | 53 | 10 | 0.58 |
| Protomata | 24 | 19 | 6 | 5 | 1 | 4.8 |
| Random Forest | 16 | 968 | 5 | 2 | 0.5 | 8 |
| Snort | 15 | 98 | 17 | 14 | 0.4 | 1.07 |
| SPM | 14 | 6631 | 5 | 0.5 | 0.1 | 28 |

## 6.5 Performance Comparison

For each of the ANMLZoo benchmarks, Table 6.9 shows the performance of competing CPU and GPU automata processing frameworks. The CPU implementation is Intel Hyperscan [1] measured independently by the authors using a 3.1 GHz Intel i5-4440 CPU with 32 GB RAM. The GPU implementation is iNFAnt2 executed on an Nvidia Titan Xp as reported in [11].

In order to understand the relationship between the NFA and its corresponding performance on the CPU and GPU implementations, the table also lists runtime data for each benchmark: the average number of active states (active set) and total



المنارة للاستشارات
www.manaraa.com

Table 6.10 Repertoire of Achieved Configurations on Stratix10 GS

| # STEs | Hardware Fan-out | Output Encoders | Max Reporting Cycles | Max Report rate (GHz) | Fmax (MHz) | Max BW for $N_{\%active} = 0.25 (GB/s)$ |
|---|---|---|---|---|---|---|
| 4K | 254 | 16 | 100% | 4.64 | 290 | 8746 |
| 8K | 126 | 32 | 50% | 8 | 250 | 7510 |
| 12K | 83 | 48 | 33% | 12 | 250 | 7331 |
| 16K | 62 | 64 | 25% | 13.4 | 210 | 6208 |
| 20K | 49 | 80 | 20% | 15.2 | 190 | 5549 |
| 24K | 40 | 96 | 17% | 16.32 | 170 | 4863 |
| 28K | 34 | 112 | 14% | 16.8 | 150 | 4255 |
| 32K | 30 | 128 | 12% | 16.64 | 130 | 3719 |
| 36K | 26 | 144 | 11% | 15.84 | 110 | 3068 |
| 40K | 23 | 160 | 10% | 14.4 | 90 | 2467 |
| 44K | 21 | 176 | 9% | 12.32 | 70 | 1744 |
| 48K | 19 | 192 | 8% | 9.6 | 50 | 1072 |

number of reports as reported in [11]. NAPOLY performs best for larger benchmarks with more active states and is faster than both the GPU and CPU NFA implementations in 10 of the 12 benchmarks, while the GPU implementation is faster in only 2 benchmarks and the CPU implementation has no winning benchmarks.

$$\frac{1}{\frac{1}{2} + \frac{1}{2} \times \frac{1}{2}} \approx 1.33 \tag{6.1}$$

## 6.6 Overlay Scalability

As shown in Equation 4.1, NAPOLY throughput depends on (1) the number of reconfigurations needed, which may be reduced by having a larger overlay with more interconnect density, (2) the time to flush the input buffer, which depends on clock speed, and (3) reconfiguration time, which depends on DRAM bandwidth. Table 6.10 shows NAPOLY capability when scaled up to an Intel Stratix 10 GS. However, even if larger FPGA can offer roughly double of of overlay capacity, double of clock



المنارة للاستشارات
www.manaraa.com

rate and double of DRAM bandwidth, the performance won't probably be doubled according to Equation 6.1.



المنارة للاستشارات
www.manaraa.com

# Chapter 7

# Conclusion and Future work

In this dissertation we have presented a novel architecture for an automata processor overlay and its associated software. NAPOLY is paramerterizable, allowing for tradeoffs in state capacity, interconnect density, and output buffer size. These tradeoffs allow for offline generation of a repertoire of overlays that allow for the overlay to be customized for specific types of NFAs. Once an overlay is deployed, the user can rapidly program the NFA at runtime, supporting arbitrary large NFAs. Automata-based benchmarks are mapped to NAPOLY processing elements based on SAT solver mappable technique.

Our performance results included the time required to program the overlay from DRAM and are competitive with the state-of-the-art CPU implementation from Intel and the state-of-the-art GPU implementation. Our performance results showed that NAPOLY's performance scales with on-chip memory capacity. In addition, we evaluated NAPOLY's scalability on larger FPGA, Stratix 10 GS.

## 7.1 Future Research Directions

There is two directions still needed to explore in this research in order to improve NAPOLY's performance. First is the number of unused hardware resources and how can be reduced. Second is to the high latency of flushing the input and output buffers and how can be eliminated.



المنارة للاستشارات
www.manaraa.com

#### 7.1.1 STE Reach versus Fan-out

The major limitation in NAPOLY implementation is the hardware fan-out, which determines the number of neighbor STEs to which each STE can connect to, and determines the maximum distance that each STE can reach. This number of wires is limited by the chip hardware resources (gates), which consequently affects the overlay size (or total STEs), and limits the performance. In this dissertation, we assumed that STE "reach" is equivalent to STE "fan-out". As shown in Table 7.1, the actual number of STE outgoing and incoming wires that are actually utilized in each benchmark, is less than 29% for both, which means that about 70% of the wires are unused.

Table 7.1 Wire Utilization Achieved For ANMLZoo Benchmarks

| Benchmark | Max Logical Fan-in/ Fan-out | Min Hardware Fan-in/ Fan-out | Average Fan-in degree | Average Fan-out degree | Fan-in Wire Utiliz | Fan-out Wire Utiliz |
|---|---|---|---|---|---|---|
| Brill | 4/4 | 8/8 | 1.11 | 0.72 | 13.8% | 9% |
| ClamAV | 11/2 | 18/18 | 1.01 | 1.003 | 5.6% | 5.6% |
| DotStar | 2/2 | 4/4 | 1.00 | 0.48 | 25% | 12% |
| Entity Resolution | 28/5 | 41/41 | 1.89 | 1.15 | 4.6% | 2.8% |
| Fermi | 2/2 | 5/5 | 1.33 | 1.41 | 26.6% | 28.2% |
| Hamming | 4/2 | 14/14 | 1.69 | 1.69 | 12% | 12% |
| Levenshtein | 8/5 | 16/16 | 2.89 | 1.63 | 18% | 10.2% |
| PowerEn | 4/3 | 6/6 | 1.08 | 0.51 | 18% | 8.5% |
| Protomata | 3/106 | 9/9 | 1.02 | 0.49 | 11.3% | 5.4% |
| Random Forest | 2/2 | 6/6 | 1.05 | 0.5 | 17.5% | 8.3% |
| Snort | 19/19 | 9/9 | 1.22 | 0.6 | 13.5% | 6.6% |
| SPM | 3/2 | 6/6 | 2.1 | 1.05 | 35% | 17.5% |

One the other hand, based on our experimental results shown in the previous chapter, such automata application benchmarks requires larger overlay in order to reduce number of times required to reconfigure the chip and improve the performance. However, this is limited by hardware fan-out. To overcome this bottleneck, we need to re-design the STE to support less number of fan-out, but further reach. This goal



المنارة للاستشارات
www.manaraa.com

can be achieved by allowing the STEs to share the wires and tri-state the connections. By this way, each STE can have less number of outgoing and incoming wires and less resource utilization, but it can reach far STE. Figure 7.1 shows an example of an overlay having its STEs sharing the wires using the multiplexers. In the example, the maximum fan-out is 2, one connecting to the self-loop and the other can connect to either one of the four next STEs. As shown, $STE0$ reaches $STE_0$ through wire #0 and $STE_0$ reaches $STE1$, $STE2$, $STE3$, and $STE4$ through the multiplexer $4x1$.

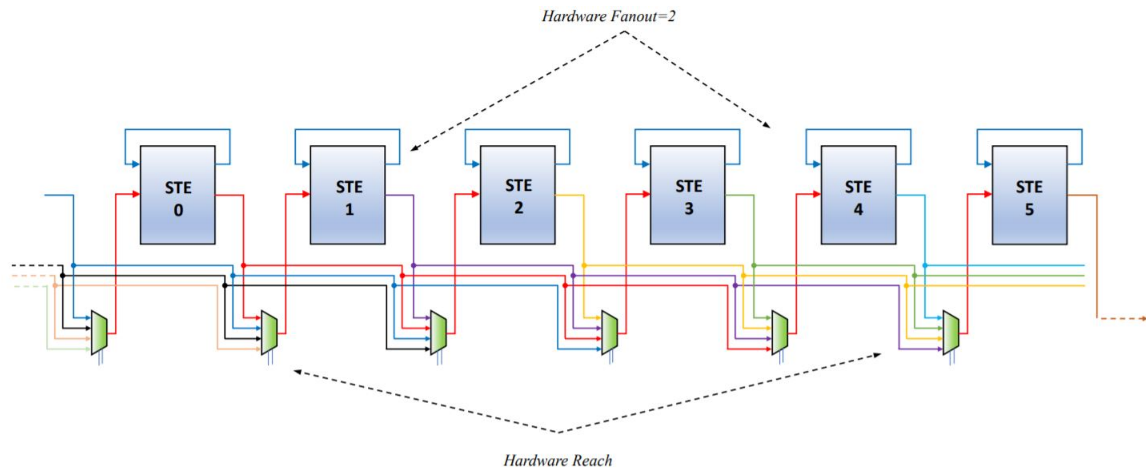



Figure 7.1 Suggested NAPOLY design

Sharing the wires costs a lot of wires, specially if the wires are implemented as a one global shared bus. If the wires are implemented hierarchically as a group of local shared wires, less number of wires can be needed, however programming its routing becomes more problematic as compared to one global bus.

### 7.1.2 NAPOLY execution time

NAPOLY spends over half of its time flushing the input buffer into the STE array and nearly half the time flushing the output buffer to DRAM. It is possible to perform these steps in parallel, if reports are written to DRAM immediately after being generated from the STE array.



المنارة للاستشارات
www.manaraa.com

The current design of NAPOLY is based on performing one operation at a time, for example, flushing the input buffer while the rest of operations (flushing output buffer, filling input buffer and filling the next-state table and gates) are stall. Since flushing the input buffer and the output buffer have the major effect on NAPOLY performance, overlapping the two operations can hide this latency. This can be implemented by splitting the overlay into two halves. While the first half is flushing the input buffer, the second half is flushing the output buffer. This can eliminate the time of flushing the input buffer of NAPOLY performance, however at cost of more reconfigurations can be required.

The time to flush the output buffer can also be reduced by adding FIFO in the report region in order to allow outputting reports while flushing the output buffer. This approach will help in eliminating the latency of output buffer flush, and potentially requiring smaller output buffer.



المنارة للاستشارات
www.manaraa.com

# Bibliography

[1] K. Angstadt and et al. Mncart: An open-source, multi-architecture automata-processing research and execution ecosystem. 2017.

[2] M. Becchi and P. Crowley. Efficient regular expression evaluation: theory to practice. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2008.

[3] Michela Becchi and Crowley Patrick. Data structures, algorithms and architectures for efficient regular expression evaluation. 2009.

[4] V. Betz and J. Rose. Vpr: a new packing, placement and routing tool for fpga research. In *Proceedings of the 1997 International Workshop on Field Programmable Logic and Applications*, 1997.

[5] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(12), 2014.

[6] A. Putnam et al. A reconfigurable fabric for accelerating large-scale datacenter services. *Computer Architecture, ACM/IEEE International Symposium on*, 2014.

[7] Chunkun Bo et al. Automata processing in reconfigurable architectures: In-the-cloud deployment, cross-platform evaluation, and fast symbol-only reconfiguration. 2019.

[8] D. Guo et al. A scalable multithreaded l7-filter design for multi-core servers. *ANCS'08*, 2008.

[9] I. Roy et al. Interval stabbing on the automata processor. *Journal of Parallel and Distributed Computing*, 2018.

[10] J. Hauswald et al. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. 2015.



المنارة للاستشارات
www.manaraa.com

[11] J. Wadden et al. Anmlzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016.

[12] J. Yu et al. Time-division multiplexing automata processor. 2019.

[13] K. Angstadt et al. Mncart: An open-source, multi-architecture automata-processing research and execution ecosystem. 2018.

[14] K. Peng et al. Chain-based dfa deflation for fast and scalable regular expression matching using tcam. 2011.

[15] K. Wang et al. Association rule mining with the micron automata processor. In *Proceedings of the IEEE 29th International Parallel and Distributed Processing Symposium*, 2015.

[16] M. Casias et al. Debugging support for pattern-matching languages and accelerators. *ASPLOS*, 2019.

[17] N. Cascarano et al. infant: Nfa pattern matching on gpgpu devices. *ACM SIGCOMM Computer Communication Review 40.5*, 2010.

[18] Q. Wang et al. A dna motif search tool using the micron automata processor and fpga. *IEICE Transactions on Information and Systems*, 2017.

[19] R. Moussalli et al. A study on parallelizing xml path filtering using accelerators. 2014.

[20] R. Nishtala et al. Scaling memcache at facebook. 2013.

[21] X. Chen et al. Picking pesky parameters: Optimizing regular expression matching in practice. *IEEE Transactions on Parallel and Distributed Systems*, 27(5), 2016.

[22] Y. Fang et al. Fast support for unstructured data processing: the unified automata processor. In *Proceedings of MICRO-48*, 2015.

[23] Y. Yang et al. Compact architecture for high-throughput regular expression matching on fpga. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2008.




المنارة للاستشارات
www.manaraa.com

[24] G. Li F. Seide and D. Yu. Conversational speech transcription using context-dependent deep neural networks. 2011.

[25] P. Flicek and E. Birney. Sense from sequence reads: methods for alignment and assembly. *Nature methods*, 2009.

[26] A. Harris. Distributed caching via memcached. 2010.

[27] Peter Linz. An introduction to formal languages and automata. 2006.

[28] M. Nourian, X. Wang, W. Feng X. Yu, and M. Becchi. Demistifying automata processing: Gpus, fpgas or micron's ap? In *Proceedings of the International Conference on Supercomputing*, 2017.

[29] I. Roy and S. Aluru. Finding motifs in biological sequences using the micron automata processor. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*, 2014.

[30] R. Sidhu and V. K. Prasanna. Fast regular expression matching using fpgas. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.

[31] Arun Subramaniyan and Reetuparna Das. Parallel automata processor. 2017.

[32] Louis Woods Teubner, Jens and Chongling Nie. Skeleton automata for fpgas: reconfiguring without reconstructing. 2012.

[33] T. Tracy, M. Stan, N. Brunelle, J. Wadden, K. Wang, K. Skadron, and G. Robins. Nondeterministic finite automata in hardware – the case of the levenshtein automaton. In *Proceedings of the 5th International Workshop on Architectures and Systems for Big Data in conjunction with the 42nd International Symposium on Computer Architecture*, 2015.

[34] T. Tracy, M. Stan, N. Brunelle, J. Wadden, K. Wang, K. Skadron, and G. Robins. Nondeterministic finite automata in hardware âĂŞ the case of the levenshtein automaton. In *Proceedings of the International Workshop on Architectures and Systems for Big Data (ASBD) in conjunction with the 42nd International Symposium on Computer Architecture (ISCA 2015)*, 2015.

[35] J. Wadden, K. Angstadt, and K. Skadron. Characterizing and mitigating output reporting bottlenecks in spatial automata processing architectures. In *Proceed-*



المنارة للاستشارات
www.manaraa.com

*ings of the 24th IEEE International Symposium on High-Performance Computer Architecture (HPCA'18)*, 2018.

[36] J. Wadden, K. Samira Khan, and K. Skadron. Automata-to-routing: An open-source toolchain for design-space exploration of spatial automata processing architectures. In *Proceedings of the IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines*, 2017.

[37] J. Wadden and K. Shadron. Vasim: An open virtual automata simulator for automata processing application and architecture research. *Technical Report CS2016-03, University of Virginia, 2016*, 2016.

[38] K. Wang, K. Angstadt, C. Bo, N. Brunelle, E. Sadredini, T. Tracy, J. Wadden, M. Stan, and K. Skadron. An overview of micron's automata processor. In *Proceedings of the 11th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2017.

[39] K. Wang and K. Skadron M. Stan. Association rule mining with the micron automata processor. In *Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium*, 2015.

[40] Y. Yang and V. Prasanna. High-performance and compact architecture for regular expression matching on fpga. *IEEE Transactions on Computers*, 61(7), 2012.

[41] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, 2006.

[42] K. Zhou, J.J. Fox, K. Wang, D.E. Brown, and K. Skadron. Brill tagging on the micron automata processor. In *Proceedings of the 9th IEEE International Conference on Semantic Computing (ICSC)*, 2015.

[43] K. Zhou, J. Wadden, J.J. Fox, K. Wang, D.E. Brown, and K. Skadron. Regular expression acceleration on the micron automata processor: Brill tagging as a case study. In *Proceedings of the IEEE International Conference on Big Data (Big Data 2015)*, 2015.



المنارة للاستشارات
www.manaraa.com

# Appendix A

# Mapping Heuristic

## A.1 validate_edges

**validate_edges** contains the top-level do-while loop, which iterates until there are no mapping violations. On each iteration, it validates the placement of each pair of states associated with each NFA edge.

For every mapping violation, **validate_edges** will evaluate the difference in score given by each of the $2 \times (f-1)$ potential resolutions, where $f$ is the hardware fan-out. In other words, for every edge comprised of predecessor state $p$ and successor state $s$, the routine can fix the violation by either remapping $s$ within the range of reachable STEs to $p$ or remapping $p$ within range of reachable STEs to $s$, where "within range" refers to any STE in the $f-1$ positions from $\lfloor \frac{f-1}{2} \rfloor$ locations less and $\lfloor \frac{f}{2} \rfloor$ greater than the target STE location. **validate_edges** eventually chooses one move that results in most positive or least negative impact on the score.

## A.2 check_move

The **check_move** A.1 routine evaluates the effect of re-mapping a state in terms of its impact on the mapping score. Re-mapping a state from its original location in STE $n$ to new location in STE $m$ where $n < m$ (i.e. moving a state to a larger STE index) will affect any edge whose predecessor or successor state is mapped to STE $l : n \leq l \leq m$, or where $m < n$ (i.e. moving a state to a lower STE index) will affect any edge whose predecessor or successor is mapped to STE $l : m \leq l \leq n$.



المنارة للاستشارات
www.manaraa.com

### A.3 Move_STE

**move_STE** performs a remapping operation on the graph by reassigning the state in STE index $from$ to STE index *to*. Moving a state in this way causes the states mapped in the range of STEs between $from$ and *to* to be shifted by one in order to fill the gap left by the state being moved.

This operation is depicted in Fig. A.1. In this example, there is an edge connecting states "fifth" and "second" that are mapped to STEs $n$ and $m$, respectively. Since $n > m$, the edge is oriented in the upward direction in the figure, in which higher-numbered STEs are lower as compared to lower-numbered STEs. The left side shows the original mapping state. Moving the state "fifth" from STE $n$ to STE $m$ causes all the states between them to shift down, as shown on the right side. This affects the mapping score contribution of any edges having successors or predecessors in the range of $n$ to $m$.



المنارة للاستشارات
www.manaraa.com

```
Function validate_edges():
    Input: NFA edges
    Output: NFA edges
    do
        for edge p → s in current STE assignment do
            // check for a mapping violation
            if ((p − s) < (−f−1)/2) || ((s − p) > f/2) then
                max_differential_score = - INT_MAX // evaluate each potential
                solution to the violation...
                for k = −⌊(f−1)/2⌋ ... ⌊f/2⌋ do
                    to = s + k
                    // ... by moving the predecessor closer to the successor
                    max_differential_score = check_move(from, to, max_differential_score,
                    best_from, best_to)
                end
                for k = −⌊(f−1)/2⌋ ... ⌊f/2⌋ do
                    from = s to = p + k
                    // ... by moving the successor closer to the predecessor
                    max_differential_score = check_move(from, to, max_differential_score,
                    best_from, best_to)
                end
                move_ste(best_from,best_to)
            end
        end
        // avoid getting suck in a local minema
        if # of violations unchanged for 10 iterations then
            make 10000 random moves
        end
    while fan-out constraint violations exist;
```



المنارة للاستشارات
www.manaraa.com

```
Function check_move():
    Input: from, to
    Output: max_differential_score, best_to, best_from
    // score for edges affected by the remapping
    score = calculate_score(from, to);
    // perform the remapping
    move_SE(from, to)
    // re-calculate score
    differential_score = score - calculate_score(from, to)
    // revert mapping to previous state
    move_SE(from, to); // undo move
    // check if the new score is better than the best found so far
    if differential_score > max_differential_score then
        max_differential_score = differential_score
        best_to = to
        best_from = from
    end
```



المنارة للاستشارات
www.manaraa.com

```
Function move_STE():
    Input: from, to
    Output: NFA edges
    if from < to then
        for edge i → j do
            if j == from then
                replace i → j with i → to
            else if j > from && j ≤ to then
                replace i → j with i → j − 1
            end
        end
    else
        for edge i → j do
            if j == from then
                replace i → j with i → to
            else if j > to && j < from then
                replace i → j with i → j + 1
            end
        end
    end
Function calculate_score():
    Input: from, to
    sum = 0
    for edge i → j such that (from ≤ i ≤ to || to ≤ i ≤ from ) || (from ≤ j ≤ to
    || to ≤ j ≤ from) do
        sum = sum + |i − j|
    end
    return sum
```



المنارة للاستشارات
www.manaraa.com

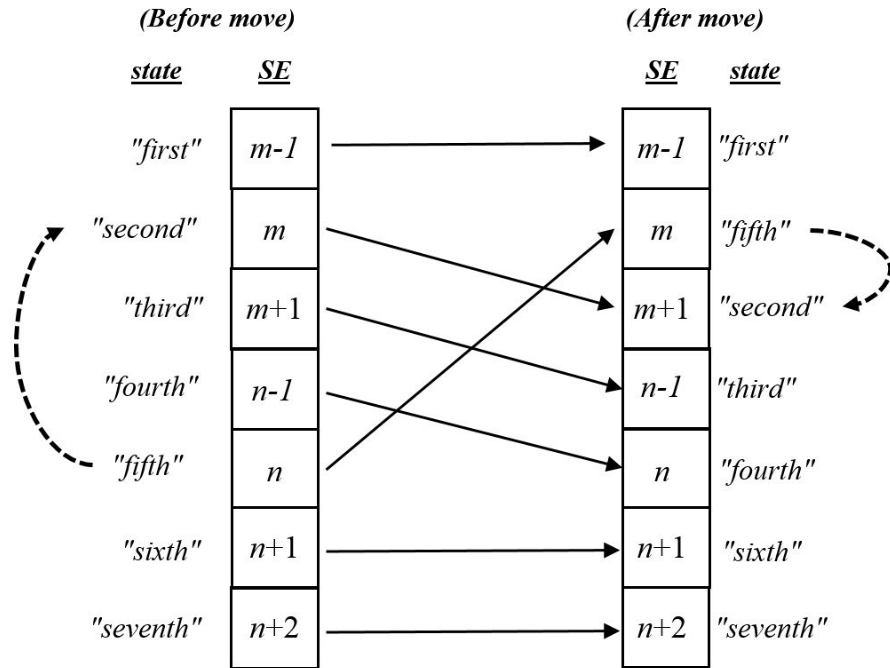



Figure A.1 Remapping STEs. Edge between state "fifth" and "second" is reassigned from STEs $n$ and $m$, where $n > m$, to $m$ and $m + 1$ (after an operation "move STE $n$ to $m$"). In this case, a movement from a higher-numbered STE to a lower-numbered STE causes all other STEs assignments between the two values to shift up, requiring an update to all other edges involving these STEs.

## A.4 CALCULATE_SCORE

**calculate_score** accumulates the "distance" of all edges having successors or predecessors mapped to any of the STEs in a given STE range, where the distance is defined as the absolute difference in STE numbers corresponding to the states that comprise the edge. The mapping heuristic's objective is to minimize this score by mapping connected STEs into localized regions in the STE array.



المنارة للاستشارات
www.manaraa.com